SystemVerilog User Guide

Version C-2009.06, June 2009

SYNOPSYS®

Copyright Notice and Proprietary Information

Right to Copy Documentation

The license agreement with Synopsys permits licensee to make copies of the documentation for its internal use only. Each copy shall include all copyrights, trademarks, service marks, and proprietary rights notices, if any. Licensee must assign sequential numbers to all copies. These copies shall contain the following legend on the cover page:

> "This document is duplicated with the permission of Synopsys, Inc., for the exclusive use of _____ and its employees. This is copy number _____."

Destination Control Statement

All technical data contained in this publication is subject to the export control laws of the United States of America. Disclosure to nationals of other countries contrary to United States law is prohibited. It is the reader's responsibility to determine the applicable regulations and to comply with them.

Disclaimer

SYNOPSYS, INC., AND ITS LICENSORS MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Registered Trademarks (®)

Synopsys, AMPS, Astro, Behavior Extracting Synthesis Technology, Cadabra, CATS, Certify, CHIPit, Design Compiler, DesignWare, Formality, HDL Analyst, HSIM, HSPICE, Identify, iN-Phase, Leda, MAST, ModelTools, NanoSim, OpenVera, PathMill, Physical Compiler, PrimeTime, SCOPE, Simply Better Results, SiVL, SNUG, SolvNet, Syndicated, Synplicity, Synplify, Synplify Pro, Synthesis Constraints Optimization Environment, TetraMAX, the Synplicity logo, UMRBus, VCS, Vera, and YIELDirector are registered trademarks of Synopsys, Inc.

Trademarks (™)

AFGen, Apollo, Astro-Rail, Astro-Xtalk, Aurora, AvanWaves, BEST, Columbia, Columbia-CE, Confirma, Cosmos, CosmosLE, CosmosScope, CRITIC, CustomSim, DC Expert, DC Professional, DC Ultra, Design Analyzer, Design Vision, DesignerHDL, DesignPower, DFTMAX, Direct Silicon Access, Discovery, Eclypse, Encore, EPIC, Galaxy, Galaxy Custom Designer, HANEX, HAPS, HapsTrak, HDL Compiler, Hercules, Hierarchical Optimization Technology, High-performance ASIC Prototyping System, HSIM^{plus}, i-Virtual Stepper, IICE, in-Sync, iN-Tandem, Jupiter, Jupiter-DP, JupiterXT, JupiterXT-ASIC, Liberty, Libra-Passport, Library Compiler, Magellan, Mars, Mars-Rail, Mars-Xtalk, Milkyway, ModelSource, Module Compiler, MultiPoint, Physical Analyst, Planet, Planet-PL, Polaris, Power Compiler, Raphael, Saturn, Scirocco, Scirocco-i, Star-RCXT, Star-SimXT, System Compiler, System Designer, Taurus, TotalRecall, TSUPREM-4, VCS Express, VCSi, VHDL Compiler, VirSim, and VMC are trademarks of Synopsys, Inc.

Service Marks (^{SM})

MAP-in, SVP Café, and TAP-in are service marks of Synopsys, Inc.

SystemC is a trademark of the Open SystemC Initiative and is used under license.
ARM and AMBA are registered trademarks of ARM Limited.
Saber is a registered trademark of SabreMark Limited Partnership and is used under license.
All other product or company names may be trademarks of their respective owners.

Contents

Preface

This preface includes the following sections:

What's New in This Release

Information about new features, enhancements, and changes, along with known problems, limitations, and resolved Synopsys Technical Action Requests (STARs), is available in the *HDL Compiler Release Notes* in SolvNet.

To see the *SystemVerilog Release Notes*,

1. Go to the Download Center on SolvNet located at the following address:

   https://solvnet.synopsys.com/DownloadCenter

   If prompted, enter your user name and password. If you do not have a Synopsys user name and password, follow the instructions to register with SolvNet.
2. Select SystemVerilog, and then select a release in the list that appears.

About This Manual

This document describes the SystemVerilog constructs currently supported by the Synopsys synthesis tool.

Audience

This document is for logic designers who are familiar with the Synopsys Design Compiler tool and the HDL Compiler (Presto Verilog) tool. Knowledge of the Verilog language is required. This document is not a stand-alone document but must be used in conjunction with the *HDL Compiler (Presto Verilog) Reference Manual* and the *SystemVerilog IEEE Std 1800-2005*.

Related Publications

You might also want to refer to the documentation for the following related Synopsys products:

- HDL Compiler (Presto Verilog)
- Formality and VCS

Conventions

The following conventions are used in Synopsys documentation.

Convention	Description
`Courier`	Indicates command syntax.
`Courier italic`	Indicates a user-defined value in Synopsys syntax, such as *`object_name`*. (A user-defined value that is not Synopsys syntax, such as a user-defined value in a Verilog or VHDL statement, is indicated by regular text font italic.)
`Courier bold`	Indicates user input—text you type verbatim—in Synopsys syntax and examples. (User input that is not Synopsys syntax, such as a user name or password you enter in a GUI, is indicated by regular text font bold.)
[ ]	Denotes optional parameters, such as `pin1 [pin2 ... pinN]`
|	Indicates a choice among alternatives, such as `low | medium | high` (This example indicates that you can enter one of three possible values for an option: low, medium, or high.)
_	Connects terms that are read as a single term by the system, such as `set_annotated_delay`
Control-c	Indicates a keyboard combination, such as holding down the Control key and pressing c.
\	Indicates a continuation of a command line.
/	Indicates levels of directory structure.
Edit > Copy	Indicates a path to a menu command, such as opening the Edit menu and choosing Copy.

Customer Support

Customer support is available through SolvNet online customer support and through contacting the Synopsys Technical Support Center.

Accessing SolvNet

SolvNet includes an electronic knowledge base of technical articles and answers to frequently asked questions about Synopsys tools. SolvNet also gives you access to a wide range of Synopsys online services including software downloads, documentation on the Web, and “Enter a Call to the Support Center.”

To access SolvNet, go to the SolvNet Web page at the following address:

https://solvnet.synopsys.com

If prompted, enter your user name and password. If you do not have a Synopsys user name and password, follow the instructions to register with SolvNet.

If you need help using SolvNet, click HELP in the top-right menu bar or in the footer.

Contacting the Synopsys Technical Support Center

If you have problems, questions, or suggestions, you can contact the Synopsys Technical Support Center in the following ways:

- Open a call to your local support center from the Web by going to https://solvnet.synopsys.com, entering your user name and password and then clicking “Enter a Call to the Support Center.”
- Send an e-mail message to your local support center.
  - E-mail support_center@synopsys.com from within North America.
  - Find other local support center e-mail addresses at http://www.synopsys.com/Support/GlobalSupportCenters/Pages
- Telephone your local support center.
  - Call (800) 245-8005 from within the continental United States.
  - Call (650) 584-4200 from Canada.
  - Find other local support center telephone numbers at http://www.synopsys.com/Support/GlobalSupportCenters/Pages

1

Introduction to SystemVerilog for Synthesis

This document describes the SystemVerilog constructs currently supported by the Synopsys synthesis tool.

The synthesizable subset of SystemVerilog currently implemented by Synopsys is listed in Table 1-1. See the IEEE Std 1800-2005 for syntax details.

This chapter contains the following sections:

- Currently Supported Constructs
- Reading SystemVerilog Files
- Reading SystemVerilog With acs_read_hdl
- Reading Large Designs
- Hierarchical Elaboration Flow for SystemVerilog Designs With Interfaces
- Naming Problems: Long Names
- Reading Designs With Assertion Checker Libraries
- Testbenches - Using the Netlist Wrapper
- Elaboration Reports

Trouble-shooting guidelines and tool limitations are described in Chapter 9, “Troubleshooting and Problem Solving” and Appendix B, “Unsupported Constructs.”

Currently Supported Constructs

Table 1-1 lists the supported SystemVerilog features and indicates if an example of feature usage is provided in this document.

Table 1-1 Currently Supported Constructs

Category	Feature	Usage information
Literals	Structure literals	"Structures" on page 8-8
	Unsized literal ('0, '1, 'x, 'z)	"Structures" on page 8-8
Data Types	Logic (4-value) data type	Used in most examples
	Integer data types (int, bit)	"Other Coding Styles" on page 8-1
	User-defined types (typedef)	"FSM: Using Enumerations Without an Explicit Base Value" on page 6-3, "Functions and Tasks" on page 8-16, "Using the typedef Construct" on page 9-23
	Structures (packed & unpacked)	"Structures" on page 8-8
	Enumerations	"FSM: Using Enumerations Without an Explicit Base Value" on page 6-3.
	Enum methods	No specific example; see SystemVerilog LRM. For restrictions, see "Methods for Enumerated Types" on page 6-8.
	Ranged enum labels	"Specify enum Ranges" on page 6-8.
	Unions (packed)	"Unions" on page 8-13
	Casting	"Casting" on page 8-23
	Void data types	No specific example; see SystemVerilog LRM
	Assignment patterns	"Assignment Patterns" on page 8-3

Table 1-1 Currently Supported Constructs (Continued)

Category	Feature	Usage information
Arrays	Packed arrays	"Multidimensional Arrays" on page 8-27, "Casting" on page 8-23
	Unpacked arrays	"Multidimensional Arrays" on page 8-27
	Array querying ($length, $left, $right, $low, $high, $increment, $dimensions)	"Multidimensional Arrays" on page 8-27
Data declaration	Scoping	No specific example; see SystemVerilog LRM
	Constants	No specific example; see SystemVerilog LRM
	Variables	Used in many examples
Operators	"." Operator	"Structures" on page 8-8
	Auto-operators: += -= ++ -- &= |= ^=	"New Operators" on page 4-6
	Wildcard equality/inequality operators (==? and !=?)	No specific example; see SystemVerilog LRM
	<<= >>= <<<= >>>=	"New Operators" on page 4-6
	inside and case-inside	No specific example; see SystemVerilog LRM
Procedural statements	unique/priority if statement	"unique if" on page 4-4, "priority if" on page 4-4
	unique/priority case, casex, casez statements	"priority case" on page 4-4
	Matching end block name	"Using Matching Block Names" on page 8-24
Processes	always_comb	"always_comb" on page 4-2
	always_latch	"Inferring Latches Using always_latch" on page 5-2

Table 1-1 Currently Supported Constructs (Continued)

Category	Feature	Usage information
	always_ff	"Inferring Flip-Flops Using always_ff" on page 5-4
Tasks & Functions	Void functions	"$unit Synthesis Restrictions" on page 2-11, "Functions and Tasks" on page 8-16
	All types as legal task/function argument types	"Functions and Tasks" on page 8-16
	All types as legal function return types	"Functions and Tasks" on page 8-16
	Return statement in functions	"Functions and Tasks" on page 8-16
	Logic default task/function argument type	"Functions and Tasks" on page 8-16
	Input default task/function argument direction	"Functions and Tasks" on page 8-16
	Binding by name	"Tasks and Functions - Binding by .name" on page 8-5
	Automatic variable initialization	"Automatic Variable Initialization" on page 8-6
	Argument binding using the .name syntax	No specific example; see SystemVerilog LRM.
Assertions	Assertions	Assertions are parsed and ignored; see Appendix B, "Unsupported Constructs" and "Reading Designs With Assertion Checker Libraries" on page 1-22."
Hierarchy	All types as legal module ports	"Multidimensional Arrays" on page 8-27,"Functions and Tasks" on page 8-16
	$unit	"$unit Usage" on page 2-2

Table 1-1 Currently Supported Constructs (Continued)

Category	Feature	Usage information
	implicit .name and .* port connections	"Implicit Instantiation of Ports" on page 8-14
Interfaces	Interface as signal container & module port replacement	"Basic Interface" on page 7-2
	Interface modports	"Interfaces With Modports" on page 7-4
	Interface ports	"Ports in Interfaces" on page 7-18
	Parameterized interfaces	"Parameterized Interfaces" on page 7-17
	Interface tasks & functions	"Interfaces With Functions and Tasks" on page 7-12
	Generic interface	For examples, see the SystemVerilog LRM.
Parameters	Default logic type	For examples, see the SystemVerilog LRM.
	Data type parameter	"Specifying Data Types for Parameters" on page 8-19
System tasks and functions	Expression Size system function ($bits)	"Casting" on page 8-23
Compiler directives	begin_keywords and end_keywords	"'begin_keywords and 'end_keywords" on page 8-4
Flow Control	For (int i=0; ….)	"Functions and Tasks" on page 8-16
	break /continue	For examples, see the SystemVerilog LRM.

Table 1-1 Currently Supported Constructs (Continued)

Category	Feature	Usage information
	do .. while	See examples in "Using do...while Loops" on page 9-22
Packages	Scope extraction using :: Wildcard inports inside modules Wildcard imports inside $unit Specific imports	"Packages" in Chapter 3

Reading SystemVerilog Files

Read SystemVerilog code into Design Compiler with any of the following commands:

- `read_sverilog { files }`

  When reading designs containing interfaces or parameterized designs, set the `hdlin_auto_save_templates` variable to true. See Example 1-1.

- `read -f sverilog { files }`

  When reading designs containing interfaces or parameterized designs, set the `hdlin_auto_save_templates` variable to true.

- `read_file -f sverilog { files }`

  When reading designs containing interfaces or parameterized designs, set the `hdlin_auto_save_templates` variable to true.

- `analyze -f sverilog { files }`
  `elaborate <topdesign>`

  See Example 1-2.

  This method is usually recommended because

  - It does recursive elaboration of the entire design so you don't need an explicit `link` command. That is, the `elaborate` command includes the functions of the `link` command.
  - For designs containing interfaces or parameterized designs, you do not need to set the `hdlin_auto_save_templates` variable to true.

In Example 1-1 and Example 1-2, the design is contained in a file named parameterized_interface.sv, and the top-level design is named top.

Example 1-1 Script With read_sverilog Command

```
set hdlin_auto_save_templates TRUE
read_sverilog  parametrized_interface.sv
current_design top
link
compile
write -f verilog -h -o gates.parametrized_interface_rd.v
quit
```

Example 1-2 Script With analyze and elaborate Commands

```
analyze -f sverilog  parametrized_interface.sv
elaborate top
compile
write -f verilog -h -o gates.parametrized_interface_an_elab.v
quit
```

Note:

If your design contains global declarations, you must read your global files before you read your specific design files. For details, see "Reading Designs That Use $unit" on page 2-3.

Automatic Detection of RTL Language From File Extensions

You can specify a file format with the `read_file` command by using the `-format` option. If you do not specify a format, `read_file` infers the format based on the file extension. If the file extension in unknown, the format is assumed to be .ddc. The file extensions in Table 1-2 are supported for automatic inference:

Table 1-2 Supported File Extensions for Automatic Inference

Format	File extensions
ddc	.ddc
db	.db, .sldb, .sdb, .db.gz, .sldb.gz, .sdb.gz
SystemVerilog	.sv, .sverilog, .sv.gz, .sverilog.gz

The supported extensions are not case sensitive. All formats except .ddc can be compressed in gzip (.gz) format.

If you specify a file format that is not supported, Design Compiler generates an error message. For example, if you specify `read_file test.vlog`, Design Compiler issues the following DDC-2 error message:

```
Error: Unable to open file 'test.vlog' for reading. (DDC-2)
```

Reading SystemVerilog With acs_read_hdl

Automated Chip Synthesis (ACS) is a tool that provides a higher level of abstraction for synthesis, including compile strategy implementation and automatic generation of compile scripts and makefiles. You can use the `acs_read_hdl` command to read SystemVerilog files. For usage details, see the *Automated Chip Synthesis User Guide* and the man page.

Use the `sverilog` format for specifying SystemVerilog format in `acs_read_hdl` command, for example

```
acs_read_hdl … -format sverilog …
```

For setting your own SystemVerilog file extensions, use the `acs_sverilog_extensions` variable, for example

```
set acs_sverilog_extensions ".sveri .sysver"
```

Note the following limitations:

- The `-update` and `-auto_update` options will not be available for SystemVerilog designs.
- No changes are required in the code for proper functioning of `acs_read_hdl`, except for the following:
  - The following RTL objects must have global, that is, design-wide, unique names: interfaces, packages, and modules.
  - Files with macros and $unit space must be included in the relevant files. Use the `include construct to do this.

Reading Large Designs

To easily read designs containing several HDL source files and libraries, use the `analyze` command with the `-vcs` option. VCS-style analyze provides better compatibility with VCS command options and makes reading large designs easier. This feature enables automatic resolution of instantiated designs by searching for the referenced designs in user-specified libraries and then loading these referenced designs. For more details, see the *HDL Compiler (Presto Verilog) Reference Manual*.

Hierarchical Elaboration Flow for SystemVerilog Designs With Interfaces

Currently, the elaboration flow is designed for top-down elaboration, which means

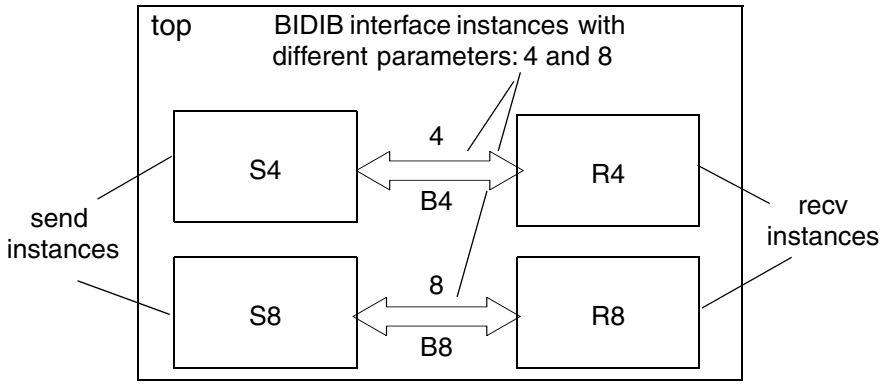
1. All RTL sources are available and analyzed
2. The top module is elaborated
3. All lower design instantiations are elaborated on demand as they are needed when the top module team instantiates them
4. The entire design is linked

The top-down flow works well, but when designs grow larger and the development teams are distributed, joining all the RTL files and elaborating the entire design can be difficult, especially when more linkage information, such as parameterized interfaces, type parameters, and user-defined types needs to be communicated between the design teams. The alternative is splitting the elaboration stage. In this scenario, every lower design team elaborates its own subdesign, saves it as a .ddc file, and later the top design team integrates the pre-elaborated modules for final linking. This is called a hierarchical elaboration flow.

The hierarchical elaboration flow is feasible with very simple nonparameterized lower designs, where no SystemVerilog interfaces are involved. However, if the lower design is complex, when the top module is elaborated, the previously elaborated lower design modules will not be recognized by the linker, and the hierarchical elaboration might fail, as shown in Figure 1-1and Example 1-3 on page 1-10, or a reelaboration will occur.

Consider the design shown in Figure 1-1. The top module in the figure contains two instances of the send module and recv module. They are connected by bidirectional bus (BIDIB) interfaces with appropriate modports and different parameters of 4 and 8.

Figure 1-1 Top Design With Send and Receive Modules Connected By Modports



The RTL in Example 1-3 describes this design. The top module instantiates two instances of the interface with different parameters: b4 (WIDTH = 4) and b8 (WIDTH = 8). The top module also instantiates two instances of send, s4 and s8, and two instances of recv, r4 and r8. The b4 BIDIB interface connects send instance s4 to recv instance r4, and the b8 interface connects send instance s8 to recv instance r8.

Example 1-3 Top Design With Send and Receive Modules Connected By Modports

```
//filename connector.sv
interface bidib;
  parameter int WIDTH=4;
  logic [WIDTH-1:0]  S;
  logic [WIDTH-1:0]  R;
  modport sendm (output S, input  R);
  modport recvm (input  S, output R);
endinterface : bidib

//filename send.sv
module send(bidib.sendm bus,
            input logic clk,
            input logic rstN );
  always_ff @(posedge clk or negedge rstN)
    if (!rstN)
      bus.S <= '0;
    else
      bus.S <= bus.R;
endmodule : send

//filename recv.sv
module recv(bidib.recvm bus);
  assign bus.R = ~ bus.S;
endmodule : recv

//filename top.sv
module top(input logic clk,
           input logic rstN);
  bidib #(.WIDTH(4)) b4();
  send s4(b4.sendm, clk, rstN);
  recv r4(b4.recvm);

  bidib #(.WIDTH(8)) b8();
  send s8(b8.sendm, clk, rstN);
  recv r8(b8.recvm);

endmodule : top
```

Assume that the design is large and complex and needs to be handled by three separate design teams: the *Send Design Team*, the *Receive Design Team*, and the *Top Design Team*.

- The *Send Design Team* synthesizes the send module by using the send.tcl script:

```
sh rm -rf ./WORK
file mkdir ./WORK
```

```
define_design_lib WORK -path ./WORK
analyze -f sverilog {connector.sv send.sv}
elaborate send
write -f ddc -hier -o send.ddc
remove_design -all
quit
```

- The *Receive Design Team* synthesizes the recv module by using the recv.tcl script:

```
sh rm -rf ./WORK
file mkdir ./WORK
define_design_lib WORK -path ./WORK
analyze -f sverilog {connector.sv recv.sv}
elaborate recv
write -f ddc -hier -o recv.ddc
remove_design -all
quit
```

- The *Top Design Team* handles the synthesis of the top module by using the top.tcl script:

```
sh rm -rf ./WORK
file mkdir ./WORK
define_design_lib WORK -path ./WORK
read_ddc send.ddc
read_ddc recv.ddc
analyze -f sverilog {connector.sv top.sv}
elaborate top
compile_ultra
write -f verilog -h -o gates.top.v
quit
```

Note:
You can modify the script to use your own compile strategy.

Because the lower designs are complex and contain interfaces, when the top module is elaborated, the previously elaborated lower designs, send.ddc and recv.ddc, will not be recognized by the linker. This results in LINK-5 warnings and LINK-3 error messages, as shown:

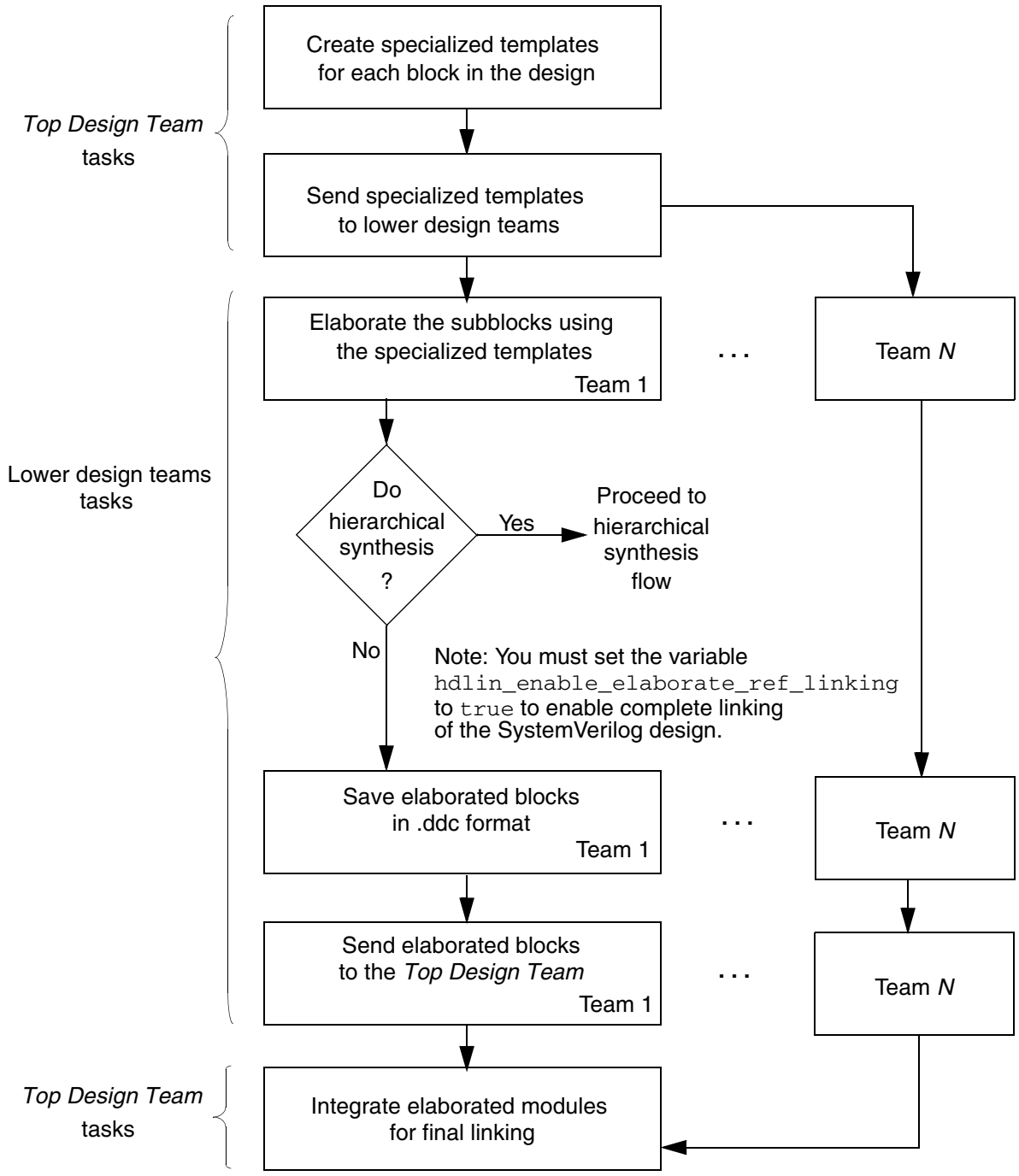
```
Error: Width mismatch on port 'bus_R' of reference to 'send' in 'top'.
(LINK-3)
Warning: Unable to resolve reference 'send' in 'top'. (LINK-5)
```

Using the Hierarchical Elaboration Flow

To avoid linking issues, use the hierarchical elaboration flow. In this flow, the *Top Design Team* creates specialized templates for each subdesign that is to be elaborated independently. These specialized templates encapsulate design context information from the top level, such as parameters, interfaces, and user-defined data types, to be used later during the elaboration of the lower design. The *Top Design Team* then passes the specialized templates to the lower block design teams. The lower design teams elaborate their subblocks by using the specialized templates and save their elaborated blocks in .ddc format. The lower design team can then send the elaborated .ddc back to the *Top Design Team* for integration or continue with the hierarchical synthesis flow. The *Top Design Team* then integrates the elaborated modules for final linking.

Figure 1-2 highlights the steps included in the hierarchical elaboration flow.

Figure 1-2 Hierarchical Elaboration Flow Steps



Methodology

Use the following procedure to enable the hierarchical elaboration flow for large designs and SystemVerilog designs that contain parameterized interfaces and modports. For example, apply the hierarchical elaboration flow to the design in Figure 1-1 on page 1-9.

Step 1: Creating the Specialized Templates

The *Top Design Team* analyzes the design and writes out the top_send and top_recv specialized templates, which contain all the linkage information between the top design and the send subdesign and between the top design and the recv subdesign. To do this automatically, you can run the top_ref.tcl script, as shown in Example 1-4 on page 1-15. Alternatively, you can perform each step manually, as shown:

1. Create and clean up your working directory, as shown:

```
sh rm -rf ./WORK
file mkdir ./WORK
define_design_lib WORK -path ./WORK
```

2. Analyze the top design with all of its interfaces:

```
analyze -f sverilog {connector.sv top.sv}
```

3. Run the `elaborate -ref` command:

```
elaborate -ref top
```

4. Run the `list_designs` command to show the specialized templates:

```
list_designs
```

   You should see a specialized template for each subdesign. For example, if you have two subdesigns, send and recv, you should see two specialized templates, top_send and top_recv, respectively.

5. Set the current design to top_send and write out the top_send template:

```
write -f ddc -hier -o top_send.ddc top_send
```

   You can then distribute the top_send template in .ddc format to the *Send Design Team*.

6. Set the current design to top_recv and write out the top_recv template:

```
write -f ddc -hier -o top_recv.ddc top_recv
```

   You can then distribute the top_recv template in .ddc format to the *Receive Design Team*.

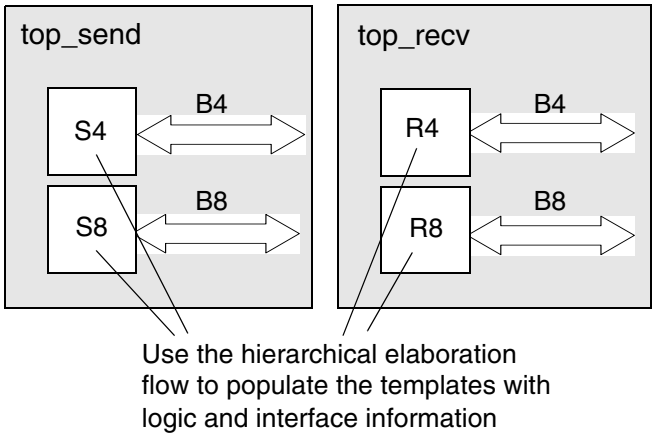
   If you receive the following LINK-5 or LBR-1 warning messages, you can ignore them:

```
Warning: Unable to resolve reference 'recv' in 'top_recv'. (LINK-5)
Warning: Can't find the design 'recv' in the library 'WORK'. (LBR-1)
```

Your current design, top_send.ddc or top_recv.ddc, has unresolved references at this point in the flow because it does not yet contain your bottom instances.

Figure 1-3 shows the specialized templates, which contain interface and connectivity information for each block in the design, such as parameterized ports, interfaces, and modports. The top_send template contains the interface and connectivity information for send blocks, and the top_recv template contains interface and connectivity information for recv blocks. The white boxes in the figure represent the area that will be populated by the logic and subblock interface and parameter information after the lower design teams elaborate the subblocks.

Figure 1-3 Specialized Templates Used For the Hierarchical Elaboration Flow



7. Clean up all temporary files and all designs in your working directory:

```
sh rm -f ./WORK/*.pvl ./WORK/*.mr ./WORK/*.syn

remove_design -all
```

To automatically analyze the design and create the specialized templates that contain all the linkage information, run the top_ref.tcl script, as shown in Example 1-4:

Example 1-4 Creating Specialized Templates Using the top_ref.tcl Script

```
sh rm -rf ./WORK
file mkdir ./WORK
define_design_lib WORK -path ./WORK
analyze -f sverilog {connector.sv top.sv}
elaborate -ref top
list_designs
write -f ddc -hier -o top_send.ddc top_send
write -f ddc -hier -o top_recv.ddc top_recv
sh rm -f ./WORK/*.pvl ./WORK/*.mr ./WORK/*.syn
remove_design -all
quit
```

Step 2: Elaborating the Subblocks and Populating the Specialized Templates

The lower design teams get specialized templates in .ddc format from the *Top Design Team*. For example, the *Send Design Team* analyzes the send module and then reads in the top_send specialized template. The specialized template includes all the linkage information, such as modports and interface parameters. The *Send Design Team* then writes out a .ddc file that contains all the send instances that were present in the top design. This creates a fully linkable send subdesign that the *Top Design Team* uses for integration.

To elaborate the subblocks and populate the specialized templates automatically, the *Send Design Team* runs the send_team.tcl script, as shown in Example 1-5 on page 1-17. Alternatively, the *Send Design Team* can perform each step manually, as shown in the following procedure:

1. Create and clean up your working directory, as shown:

```
sh rm -rf ./WORK
file mkdir ./WORK
define_design_lib WORK -path ./WORK
```

2. Set the following variable to include additional hierarchical linking information:

```
set hdlin_enable_elaborate_ref_linking true
```

3. Analyze the send subdesign with all its interfaces:

```
analyze -f sverilog {connector.sv send.sv}
```

4. Read in the top_send specialized template sent by the *Top Design Team*:

```
read_ddc top_send.ddc
```

   This provides all the linkage information such as modports, parameters in the interface, and all the send instances inside the top design.

5. Set the current design to the top_send specialized template:

```
set current_design top_send
```

6. Run the `link` command to link the interface information to the send design:

```
link
```

7. Run the `list_designs` command to list all the module names that are generated by Design Compiler:

```
list_designs
```

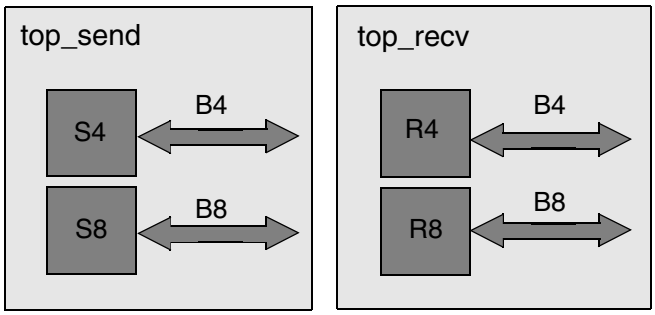
   You can use the list of designs to populate the send.ddc file with all the send design instances with linkage information. It is important that you include all the module names.

8. Write out the fully linkable send.ddc file that contains all the interface and modport information and the logic inside each module. You must send the send.ddc file to the *Top Design Team* for full-chip integration:

```
write -f ddc -hier -o send.ddc { send_I_bus_bidib_sendm_WIDTH_8
send_I_bus_bidib_sendm_WIDTH_4 }
```

Figure 1-4 shows the fully linked send and recv subdesigns with logic and interface information.

Figure 1-4 send and recv Subdesigns Populated With Logic and Interface Information



9. Clean up all temporary files and all designs in your working directory:

```
sh rm -f ./WORK/*.pvl ./WORK/*.mr ./WORK/*.syn

remove_design -all
```

You must repeat the steps for each lower-level subdesign team, write out a fully linkable .ddc file for each subdesign, and then send all the .ddc files to the *Top Design Team*. For example, if you have *Team1*, *Team2*, ... *Team-N*, the *Top Design Team* should receive the following files: *Team1*.ddc, *Team2*.ddc, ... *Team-N*.ddc.

To automatically elaborate the subblocks and populate the specialized templates that are shown in Figure 1-4, the *Send Design Team* can run the send_team.tcl script shown in Example 1-5:

Example 1-5 Elaborating Subblocks and Populating Specialized Templates Using the send_team.tcl Script

```
sh rm -rf ./WORK
file mkdir ./WORK
define_design_lib WORK -path ./WORK
set hdlin_enable_elaborate_ref_linking true
analyze -f sverilog { connector.sv send.sv }
read_ddc top_send.ddc
set current_design top_send
link
list_designs
write -f ddc -hier -o send.ddc { send_I_bus_bidib_sendm_WIDTH_8 \
 send_I_bus_bidib_sendm_WIDTH_4 }
sh rm -f ./WORK/*.pvl ./WORK/*.mr ./WORK/*.syn
remove_design -all
```

```
quit
```

The *Receive Design Team* also performs the procedure in "Step 2: Elaborating the Subblocks and Populating the Specialized Templates" on page 1-16 to manually elaborate the subblocks and populate the specialized templates that are shown in Figure 1-4. Alternatively, the *Receive Design Team* can run the recv_team.tcl script shown in Example 1-6:

Example 1-6 Elaborating Subblocks and Populating Specialized Templates Using the recv_team.tcl Script

```
sh rm -rf ./WORK
file mkdir ./WORK
define_design_lib WORK -path ./WORK
set hdlin_enable_elaborate_ref_linking true
analyze -f sverilog {connector.sv recv.sv}
read_ddc top_recv.ddc
set current_design top_recv
link
list_designs
write -f ddc -hier -o recv.ddc { recv_I_bus_bidib_recvm_WIDTH_8 \
 recv_I_bus_bidib_recvm_WIDTH_4 }
sh rm -f ./WORK/*.pvl ./WORK/*.mr ./WORK/*.syn
remove_design -all
quit
```

Step 3: Integrating the Subblocks With the Top-Level Design

The *Top Design Team* receives the send.ddc and recv.ddc designs from the *Send Design Team* and the *Receive Design Team*. First, these are read into Design Compiler. Then, the top-level design is analyzed with all the interfaces. Lastly, the design is compiled to the gate-level netlist. You should not see any LINK-3 errors or LINK-5 warnings.

To integrate the subblocks with the top-level design automatically, the *Top Design Team* can run the top_integration.tcl script, as shown in Example 1-7 on page 1-19. Alternatively, the *Top Design Team* can perform each step manually, as shown:

1. Create and clean up your working directory:

   ```
   sh rm -rf ./WORK
   file mkdir ./WORK
   define_design_lib WORK -path ./WORK
   ```

2. Set the following variable to enable the linker to review the contextual hierarchical linkage information:

   ```
   set hdlin_enable_elaborate_ref_linking true
   ```

3. Read in the fully linkable send.ddc and rec.ddc designs that were sent by the *Send Design Team* and the *Receive Design Team*, respectively:

   ```
   read_ddc send.ddc
   read_ddc recv.ddc
   ```

You should not see any LINK-3 errors or LINK-5 warnings because you are reading the fully resolved .ddc designs.

4. Analyze the top design with all the interfaces:

```
analyze -f sverilog {connector.sv top.sv}
```

5. Elaborate the top design:

```
elaborate top
```

6. Use your own compile strategy. See “Bottom-Up Compile” in the “Optimizing the Design” chapter in the *Design Compiler User Guide* for information about the hierarchical synthesis flow and additional compile strategies.

7. Write out the gate-level netlist:

```
write -f verilog -h -o gates.top.v
```

To automatically integrate the subblocks with the top-level design, run the top_integration.tcl script, as shown in Example 1-7:

Example 1-7 Integrating Subblocks With Top-Level Design Using the top_integration.tcl Script

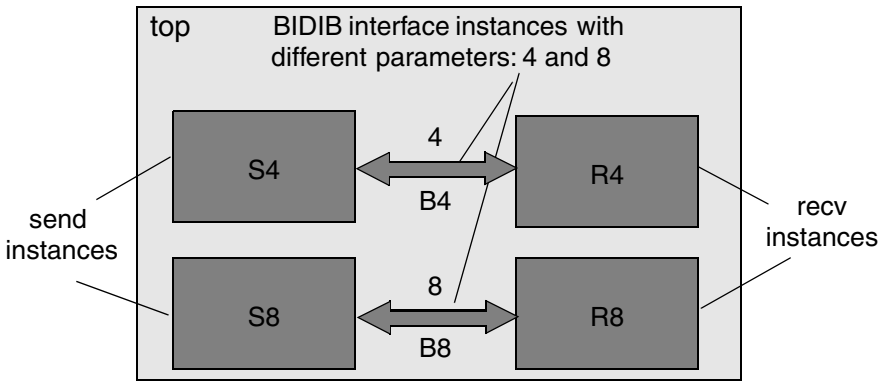
```
sh rm -rf ./WORK
file mkdir ./WORK
define_design_lib WORK -path ./WORK
set hdlin_enable_elaborate_ref_linking true
read_ddc send.ddc
read_ddc recv.ddc
analyze -f sverilog {connector.sv top.sv}
elaborate top
compile_ultra
write -f verilog -h -o gates.top.v
quit
```

Note:

You can modify the script to use your own compile strategy.

Now that the subdesigns are fully linked with all the contextual linkage information, as shown in Figure 1-5, the tool should not issue any link errors.

Figure 1-5 Top Design With Fully Linked Subdesigns



Using Compiled Lower-Level Blocks Instead of Elaborated Lower-Level Blocks

If you want to use compiled lower-level blocks instead of elaborated lower-level blocks in the top-level integration, you need to compile each instance of the lower-level block and then write out the lower-level compiled .ddc design. For example, in the design in Figure 1-1 on page 1-9, the *Send Design Team* would need to set the `current_design` to every instance of send inside top, run `compile_ultra`, and then write out the gates.send.ddc file as shown in Example 1-8 when they perform "Step 2: Elaborating the Subblocks and Populating the Specialized Templates" on page 1-16.

Example 1-8 Using Compiled Lower-Level Blocks Instead of Elaborated Lower-Level Blocks

```
set current_design send_I_bus_bidib_sendm_WIDTH_8
compile_ultra
set current_design send_I_bus_bidib_sendm_WIDTH_4
compile_ultra
write -f ddc -hier -o gates.send.ddc { send_I_bus_bidib_sendm_WIDTH_8
     send_I_bus_bidib_sendm_WIDTH_4 }
```

Then, the *Top Design Team* would use the gates.send.ddc instead of the send.ddc for the top-level integration when they perform "Step 3: Integrating the Subblocks With the Top-Level Design" on page 1-18.

Note:

Use your own compile strategy. See "Bottom-Up Compile" in the "Optimizing the Design" chapter in the *Design Compiler User Guide* for information about the hierarchical synthesis flow and additional compile strategies.

Limitations

The hierarchical elaboration flow has the following limitations:

- The hierarchical elaboration flow does not support SystemVerilog packages.
- The .* port connection style is not supported.
- The defparam usage has been deprecated in the SystemVerilog LRM and is not supported.
- You must write out the specialized templates in .ddc format (top_send.ddc and top_recv.ddc). Do not use Verilog format (top_send.v and top_recv.v). The hierarchical elaboration flow is not supported for specialized templates written in Verilog format.

Naming Problems: Long Names

If your design contains numerous interfaces and parameters, the tool creates relatively long module names due to inlining in the netlist. These long names cannot be read by some back-end tools. To shorten the name, set the `hdlin_shorten_long_module_name` to true and set `hdlin_module_name_limit` to the maximum number of characters you want in the name. When the tool builds a module name that is longer than the value specified by `hdlin_module_name_limit`, the tool will rename the module to the original name plus a hash of the full name and issue a warning indicating that the module was renamed. Example 1-9 shows the RTL module name. Example 1-10 shows the default netlist name. Example 1-11 shows the variables used to shorten the name. Example 1-12 shows the shortened name. Example 1-13 shows the warning message that the tool issues when a name is shortened.

Example 1-9 Module Name in RTL

```
module sender
```

Example 1-10 Module Name in Gate-Level Netlist Before Shortening

```
module
sender_I_i_s1_i_sendmode_I_i_s2_i_sendmode_I_i_s3_i_sendmode_I_i_s4_i_sen
dmod
e_I_i_s5_i_sendmode_I_i_s6_i_sendmode_I_i_s7_i_sendmode_I_i_s8_i_sendmode
_I_i
_s9_i_sendmode_I_i_s10_i_sendmode_
```

Example 1-11 Set Variables to Shorten

```
# Enables shortening of names
set hdlin_shorten_long_module_name true
# Specify minimum number of characters. Default: 256
```

```
set hdlin_module_name_limit 100
```

Example 1-12 Module Name in Gate-Level Netlist After Shortening

```
module sender_h_948_242_781
```

Example 1-13 Notification of Shortening

```
dc_shell-xg-t> Warning:
Design'sender_I_i_s1_i_sendmode_I_i_s2_i_sendmode_I_i_s3_i_sendmode_I_i_s
4_i_
sendmode_I_i_s5_i_sendmode_I_i_s6_i_sendmode_I_i_s7_i_sendmode_I_i_s8_i_s
endm
ode_I_i_s9_i_sendmode_I_i_s10_i_sendmode_'
was renamed to 'sender_h_948_242_781' to resolve a long name which is not
supported
by some down stream tools. (LINK-26)
```

Reading Designs With Assertion Checker Libraries

The Synopsys simulation tool, VCS, has developed numerous assertion checker libraries. Checker libraries are ignored by the synthesis tool, resulting in no hardware in the gate-level netlist.

Note:
To use the checker libraries, you must set up the search path to the library.

To understand how to read designs that use these libraries, consider the script in Example 1-14, which reads two designs that use the assert_one_hot and assert_one_cold checker libraries.

Example 1-14 Reading a Design With Checker Libraries

```
set search_path [concat $search_path /vcs7.1.1/packages/sva]

analyze -define SVA_CHECKER_INTERFACE -f sverilog {sva.inc
traffic_hot.sv}
analyze -define SVA_CHECKER_INTERFACE -f sverilog {sva.inc
traffic_cold.sv}
analyze -f sverilog {top.sv}
elaborate top
quit
```

The design files used in Example 1-14 are

- sva.inc, shown in Example 1-15 on page 1-23
- traffic_cold.sv, shown in Example 1-16 on page 1-24
- traffic_hot.sv, shown in Example 1-17 on page 1-27

- top.sv, shown in Example 1-18 on page 1-30

When reading designs containing checker libraries, use the following methodology:

1. Include the location of the checker library files in your design search path.

   In Example 1-14, the checker libraries are located in the /vcs7.1.1/packages/sva directory, so the search path is

   ```
   set search_path [concat $search_path /vcs7.1.1/packages/
   sva]
   ```

2. Create a file that includes all the checker libraries. Put this file in your design directory.

   By default, VCS provides checker libraries in individual files. For example, the assert_bits checker library is contained in the assert_bits.v file. In Example 1-14, all checker libraries are included in the sva.inc file.

3. Using the predefined macro, SVA_CHECKER_INTERFACE, analyze the checker libraries before analyzing any other module that uses the checker library.

   In Example 1-14, the commands are as follows:

   ```
   analyze -define SVA_CHECKER_INTERFACE -f sverilog {sva.inc
   traffic_hot.sv}
   analyze -define SVA_CHECKER_INTERFACE -f sverilog {sva.inc
   traffic_cold.sv}
   ```

4. Analyze the top-level module and elaborate at the top level.

   In Example 1-14, the commands are as follows:

   ```
   analyze -f sverilog {top.sv}
   elaborate top
   ```

The examples that follow contain the files used in the methodology example.

Example 1-15 Checker Libraries Contained in the sva.inc File

```
//*************sva.inc***********
`include "assert_always_on_edge.v"
`include "assert_arbiter.v"
`include "assert_bits.v"
`include "assert_change.v"
`include "assert_code_distance.v"
`include "assert_cycle_sequence.v"
`include "assert_data_used.v"
`include "assert_decrement.v"
`include "assert_delta.v"
`include "assert_driven.v"
`include "assert_dual_clk_fifo.v"
`include "assert_even_parity.v"
`include "assert_fifo.v"
`include "assert_fifo_index.v"
```

```
`include "assert_frame.v"
`include "assert_handshake.v"
`include "assert_hold_value.v"
`include "assert_implication.v"
`include "assert_increment.v"
`include "assert_memory_async.v"
`include "assert_memory_sync.v"
`include "assert_multiport_fifo.v"
`include "assert_mutex.v"
`include "assert_never.v"
`include "assert_next.v"
`include "assert_next_state.v"
`include "assert_no_contention.v"
`include "assert_no_overflow.v"
`include "assert_no_transition.v"
`include "assert_no_underflow.v"
`include "assert_odd_parity.v"
`include "assert_one_cold.v"
`include "assert_one_hot.v"
`include "assert_proposition.v"
`include "assert_quiescent_state.v"
`include "assert_range.v"
`include "assert_reg_loaded.v"
`include "assert_req_ack_unique.v"
`include "assert_stack.v"
`include "assert_time.v"
`include "assert_transition.v"
`include "assert_unchange.v"
`include "assert_valid_id.v"
`include "assert_width.v"
`include "assert_win_change.v"
`include "assert_win_unchange.v"
`include "assert_window.v"
`include "assert_zero_one_hot.v"
//*************sva.inc***********
```

Example 1-16 Module traffic_cold

```
//************traffic_cold.sv*************
module traffic_cold(output logic g0, y0, r0, g1, y1, r1, input clk, rst);

logic [4:0] ps0, ps1, ns0, ns1;

// state machine is one cold
localparam [4:0]  RST = 5'b11110,
                  SR0 =  5'b11101,
                  SR1 =  5'b11011,
                  SY  =  5'b10111,
                  SG  =  5'b01111;
initial $monitor($time,,ps0,,ps1);
// state registers
```

```
always_ff@(posedge clk or posedge rst)
begin

if (rst == 1'b1)
   begin
      ps0 <= RST;
      ps1 <= RST;
   end

else
   begin
      ps0 <= ns0;
      ps1 <= ns1;
   end

end



assert_one_cold #(0,5,0) ps0_one_cold (clk,!rst,ps0);
assert_one_cold #(0,5,0) ps1_one_cold (clk,!rst,ps1);

// next state decode
always_latch
begin
   case(ps0)
      RST :
         ns0 <= SR0;
      SR0 :
         ns0 <= SR1;
      SR1 :
         ns0 <= SG;
      SG :
         ns0 <= SY;
      SY :
         ns0 <= SR0;
   endcase

end
always_latch
begin
   case(ps1)
      RST :
         ns1 <= SG;
      SG :
         ns1 <= SY;
      SY :
         ns1 <= SR0;
      SR0 :
         ns1 <= SR1;
      SR1 :
         ns1 <= SG;
   endcase
```

```
end


// output logic

always_latch
begin
   case(ps0)
      RST :
      begin
         g0 <= 0;
         y0 <= 0;
         r0 <= 1;
      end

      SR0, SR1 :
      begin
         g0 <= 0;
         y0 <= 0;
         r0 <= 1;
      end

      SG :
      begin
         g0 <= 1;
         y0 <= 0;
         r0 <= 0;
      end

      SY :
      begin
         g0 <= 0;
         y0 <= 1;
         r0 <= 0;
      end

   endcase

end
always_latch
begin
   case(ps1)
      RST :
      begin
         g1 <= 0;
         y1 <= 0;
         r1 <= 1;
      end

      SR0, SR1 :
      begin
         g1 <= 0;
```

```
            y1 <= 0;
            r1 <= 1;
         end

         SG :
         begin
            g1 <= 1;
            y1 <= 0;
            r1 <= 0;
         end

         SY :
         begin
            g1 <= 0;
            y1 <= 1;
            r1 <= 0;
         end

      endcase

   end

   wire  e_both_yellow = y0 && y1;
   sequence e_one_yellow;
   @(posedge clk) !e_both_yellow;
   endsequence
   c_one_yellow : assert property(e_one_yellow );

   endmodule
   //************traffic_cold.sv*************
```

Example 1-17 Module traffic_hot

```
   //************traffic_hot.sv*************
   module traffic_hot(output logic g0, y0, r0, g1, y1, r1, input clk, rst);

   logic [4:0] ps0, ps1, ns0, ns1;

   // state machine is one hot
   localparam [4:0]  RST = 5'b00001,
                     SR0 =  5'b00010,
                     SR1 =  5'b00100,
                     SY  =  5'b01000,
                     SG  =  5'b10000;
   initial $monitor($time,,ps0,,ps1);

   // state registers
   always_ff @(posedge clk or posedge rst)
   begin

   if (rst == 1'b1)
      begin
         ps0 <= RST;
```

```
      ps1 <= RST;
   end

else
   begin
      ps0 <= ns0;
      ps1 <= ns1;
   end

end


assert_one_hot #(0,5,0) ps0_one_hot (clk,!rst,ps0);
assert_one_hot #(0,5,0) ps1_one_hot (clk,!rst,ps1);

// next state decode

always_latch
begin
   case(ps0)
      RST :
         ns0 <= SR0;
      SR0 :
         ns0 <= SR1;
      SR1 :
         ns0 <= SG;
      SG :
         ns0 <= SY;
      SY :
         ns0 <= SR0;
   endcase

end
always_latch
begin
   case(ps1)
      RST :
         ns1 <= SG;
      SG :
         ns1 <= SY;
      SY :
         ns1 <= SR0;
      SR0 :
         ns1 <= SR1;
      SR1 :
         ns1 <= SG;
   endcase

end


// output logic
```

```
always_latch
begin
   case(ps0)
      RST :
      begin
         g0 <= 0;
         y0 <= 0;
         r0 <= 1;
      end

      SR0, SR1 :
      begin
         g0 <= 0;
         y0 <= 0;
         r0 <= 1;
      end

      SG :
      begin
         g0 <= 1;
         y0 <= 0;
         r0 <= 0;
      end

      SY :
      begin
         g0 <= 0;
         y0 <= 1;
         r0 <= 0;
      end

   endcase

end
always_latch
begin
   case(ps1)
      RST :
      begin
         g1 <= 0;
         y1 <= 0;
         r1 <= 1;
      end

      SR0, SR1 :
      begin
         g1 <= 0;
         y1 <= 0;
         r1 <= 1;
      end

      SG :
      begin
```

```
            g1 <= 1;
            y1 <= 0;
            r1 <= 0;
         end

         SY :
         begin
            g1 <= 0;
            y1 <= 1;
            r1 <= 0;
         end

      endcase

   end

   wire  e_both_yellow = y0 && y1;
   sequence e_one_yellow;
   @(posedge clk) !e_both_yellow;
   endsequence
   c_one_yellow : assert property(e_one_yellow );

   endmodule
   //************traffic_hot.sv*************
```

Example 1-18 Module top

```
//************top.sv*************
module top(input logic clk, rst, output logic [1:0] g0, y0, r0, g1, y1,
r1);

traffic_cold  tc(.g0(g0[1]), .y0(y0[1]), .r0(r0[1]), .g1(g1[1]),
.y1(y1[1]),
.r1(r1[1]), .*);

traffic_hot  th(.g0(g0[0]), .y0(y0[0]), .r0(r0[0]), .g1(g1[0]),
.y1(y1[0]),
.r1(r1[0]), .*);

endmodule : top
//************top.sv*************
```

Testbenches - Using the Netlist Wrapper

In general, a testbench that has been written for a SystemVerilog design cannot be directly applied to the gate-level netlist because the number of ports and their types and names are not preserved in the Verilog implementation. For example, in Verilog, there are no interface type ports, parameter types, unpacked arrays or structs, or enums. Additionally, each item in a SystemVerilog interface modport is implemented in Design Compiler as a separate port. Because back-end tools can interpret only the Verilog language, you can write out only a Verilog netlist.

To make the original testbench work for a gate-level Verilog simulation, you must modify it to translate the values on the SystemVerilog ports into values on the implementation ports. The `write -f svsim` command automates this process and enables you to write out a SystemVerilog netlist wrapper for your netlist, thereby insulating users from changes in how Design Compiler translates SystemVerilog ports to Verilog ports.

The SystemVerilog netlist wrapper is a SystemVerilog module declaration. It must be instantiated by the testbench exactly as it instantiated the original SystemVerilog module declaration to create the SystemVerilog-level design under test (DUT), yielding a SystemVerilog design instance that can be driven by the testbench exactly as it drove the SystemVerilog-level DUT.

Create a Netlist Wrapper

This section describes how to write out a wrapper and gate-level netlist for a DUT that uses an interface-based design module with overridden parameter types. The test case is shown in Example 1-19.

In the test case, the module declaration TOP uses the generic interface with modport ‘i’ as a port. The dummy_top module has an instance of TOP where the interface instance IFC is overridden with the new type ‘MY_T’. The default type is ‘byte’ provided in the interface declaration IFC.

The `ifndef GATES statement defines whether you do RTL simulation or gate-level simulation. Within this block you include the wrapper file, netlist_wrapper.sv, along with the appropriate netlist or RTL file. Example 1-19 uses the netlist file, compiled_gates.v.

Example 1-19

```
typedef logic MY_T[0:2];


interface IFC #(parameter type T = byte);
  T x, y;
  modport mp (input x, output y);
endinterface
```

```
`ifndef GATES

   module TOP #(parameter type T = shortint) (interface.mp i);
     T temp;
     assign temp = i.x;
     assign i.y = temp;
   endmodule

`else


   `include "netlist_wrapper.sv"
   `include "compiled_gates.v"

`endif



`define DUT(mod) \
      `ifndef GATES \
         mod \
      `else \
         mod``_svsim \
      `endif


module dummy_top #(parameter type T = MY_T) (input T in, output T out);
         IFC#(.T(T)) ifc();
          assign ifc.x = in;
  assign out = ifc.y;

   `DUT(TOP) #(.T(T)) top_inst(ifc.mp);

endmodule
```

The above coding model is recommended when the name of the DUT is obtained from the SystemVerilog macro `DUT:

```
`DUT(TOP) #(.T(T)) top_inst(ifc.mp);
```

The `DUT definition is shown below:

```
`define DUT(mod) \
      `ifndef GATES \
         mod \
      `else \
         mod``_svsim \
      `endif
```

If the testbench is driving stimulus vectors and observing response vectors from dummy_top, the RTL simulation contains TOP as the DUT. If it is a gate-level simulation, dummy_top contains TOP_svsim. This user model makes it easier for designers to switch between RTL simulation and gate-level simulation depending on whether GATES is defined or not.

Enhancements to the Write Command

The user interface for the Design Compiler `write` command has been extended to support a svsim format for writing out the netlist wrapper. The `write -f svsim` command writes out only the netlist wrapper, not the gate-level DUT itself. To write out the gate-level DUT, you use the existing `write -f verilog` command.

Sample Script

This section describes a Design Compiler Tcl script that runs the test case.

Example 1-20

```
proc get_design_from_inst { inst } {
   return [get_attribute [get_cells $inst] ref_name]
}
analyze -f sverilog testbench.netlist.wrapper_create.netlist.wrapper.sv
elaborate dummy_top
compile

set dut [get_design_from_inst top_inst]
write -f verilog -h -o compiled_gates.v $dut
write -f svsim -o netlist_wrapper.sv $dut

quit
```

In SystemVerilog, the names of module declarations change and will depend on the interface types, modports, parameter types, parameters, and so on. Because you know the instance name that is in the testbench (or, in this case, top_inst within dummy_top) you can retrieve the cell name (the instance) from the reference name (design declaration), with the following Tcl procedure:

```
proc get_design_from_inst { inst } {
 return [get_attribute [get_cells $inst] ref_name]
}
```

After you compile your design, set the `dut` variable to the top-level instance by calling the `get_design_from_inst` procedure. Finally, write out the gate-level netlist and wrapper file, as shown:

```
set dut [get_design_from_inst top_inst]
write -f verilog -h -o compiled_gates.v $dut
write -f svsim -o netlist_wrapper.sv $dut
```

By using this methodology, you do not need to know the renamed version of the TOP module.

This methodology enables you to write the wrapper and the gate-level netlist for a specific instance of TOP, with uniquely redefined parameters, parameter types, interface types, or user-defined types within the dummy_top module, which is being driven by the testbench.

Note:

You must have the module dummy_top (which passes all of the types to the DUT) so that you can explicitly write a wrapper and gate-level netlist for this specific instance of the DUT.

Sample Wrapper File

This section describes the netlist_wrapper.sv SystemVerilog wrapper file in Example 1-21, generated by the `write -f svsim` command.

Example 1-21 Wrapper File

```
write -f svsim -o netlist_wrapper.sv $dut

`ifndef SYNTHESIS

//
// This is an automatically generated file from
// dc_shell Version B-2008.09-SP1 -- Oct 17, 2008
//

// For simulation only. Do not modify.
module TOP_svsim #(parameter type T = shortint) (interface.mp i);

      TOP_I_i_IFC_mp_T_array_1_0_2_logic_DQLcWqa_
 TOP_I_i_IFC_mp_T_array_1_0_2_logic_DQLcWqa_(
        {>>{ i.x }}, {>>{ i.y }} );
endmodule
`endif
```

The netlist wrapper contains a single module declaration called TOP_svsim, which is almost the same as that of the original SystemVerilog module TOP that was instantiated as a DUT, but with the _svsim suffix. The body of this new module is a mapping between the original

SystemVerilog ports and the Verilog implementation ports using the streaming operator >> in SystemVerilog. It contains an instance of the gate-level netlist created for the DUT. In this example, it contains an instance of

```
TOP_I_i_IFC_mp_T_array_1_0_2_logic_DQLcWqa_
```

which is the compiled_gates.v gate-level netlist file shown in Example 1-22.

Example 1-22 Gate-Level Netlist

```
module TOP_I_i_IFC_mp_T_array_1_0_2_logic_DQLcWqa_ ( \i.x , \i.y  );
  input [0:2] \i.x ;
  output [0:2] \i.y ;
  assign \i.y  [0] = \i.x  [0];
  assign \i.y  [1] = \i.x  [1];
  assign \i.y  [2] = \i.x  [2];

endmodule
```

For this mapping, the netlist wrapper must be instantiated in exactly the same way as the original DUT (that is, with the same parameter overrides and interface references, including modports). The mapping is expressed as a port connection, in positional notation, to an instance of the new Verilog DUT. Accordingly, although the RTL description of TOP had the type shortint, the instance of TOP, within dummy_top, has overridden the parameter types to MY_T. This results in the gate-level nestlist having \i.x \i.y as [0:2].

Also, each wrapper file has a version string for the version of Design Compiler that created the wrapper. In this case, it is dc_shell version B-2008.09-SP1 -- Oct 17, 2008.

Important:

The wrapper is bracketed with `ifndef SYNTHESIS because it is used only during simulation; it is not used during synthesis.

Wrapper Limitations

1. You cannot use the `-h` option with `write -f svsim`, unlike other write commands.
2. If you attempt to write out netlist wrappers for multiple designs with a single write command, Design Compiler issues an error. For example, you cannot use the `write -f svsim [list dut1 dut2]` command.
3. Only a subset of synthesizable SystemVerilog designs are supported, namely, those whose root modules use “ANSI-style” port declarations. It is recommended that all SystemVerilog code use ANSI-style port declarations. Note that the tool requires you to use the ANSI-style for all interface-type port declarations. For details on this and other tool limitations, see Appendix B, “Unsupported Constructs.”) Also note that the LRM restricts generic interface-type port declarations to only the ANSI-style. The root module in this context means the top-level module as seen by the project. If you are writing out

the top-level module of the design, then the root module and the design's top module are the same. If you are writing out a lower level module, then it is the root module that is different from the design's top-level module.

4. The netlist wrapper file cannot be read back into Design Compiler.
5. There can only be one DUT. You cannot have multiple DUTs driven by the same testbench. That is, Figure 1-6 is allowed, but Figure 1-7 is not allowed.

Figure 1-6 Single DUT Testing Allowed

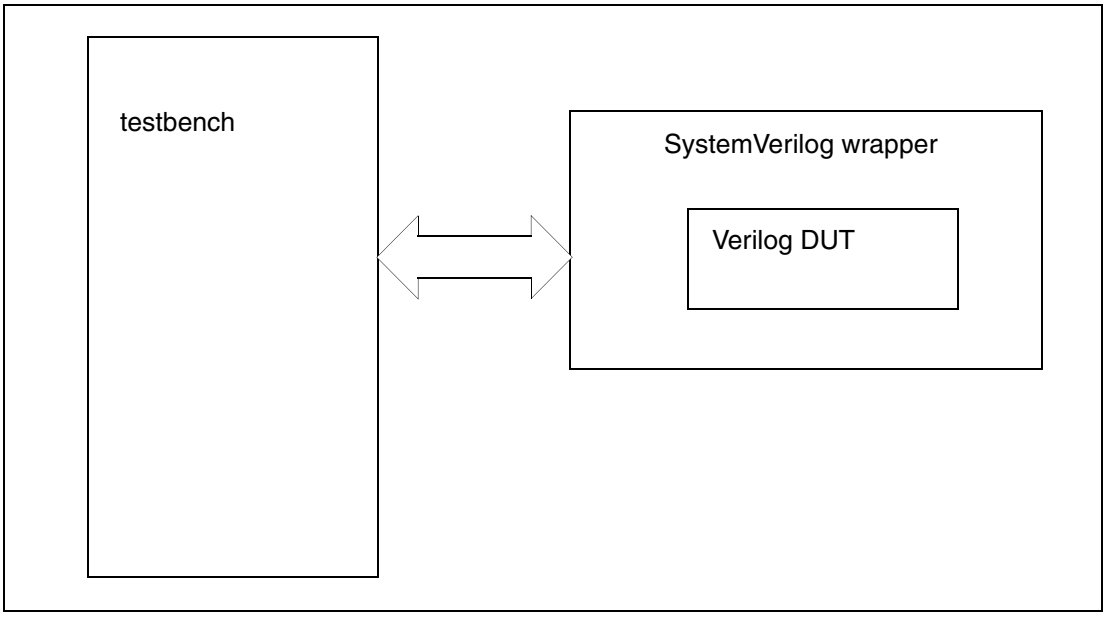
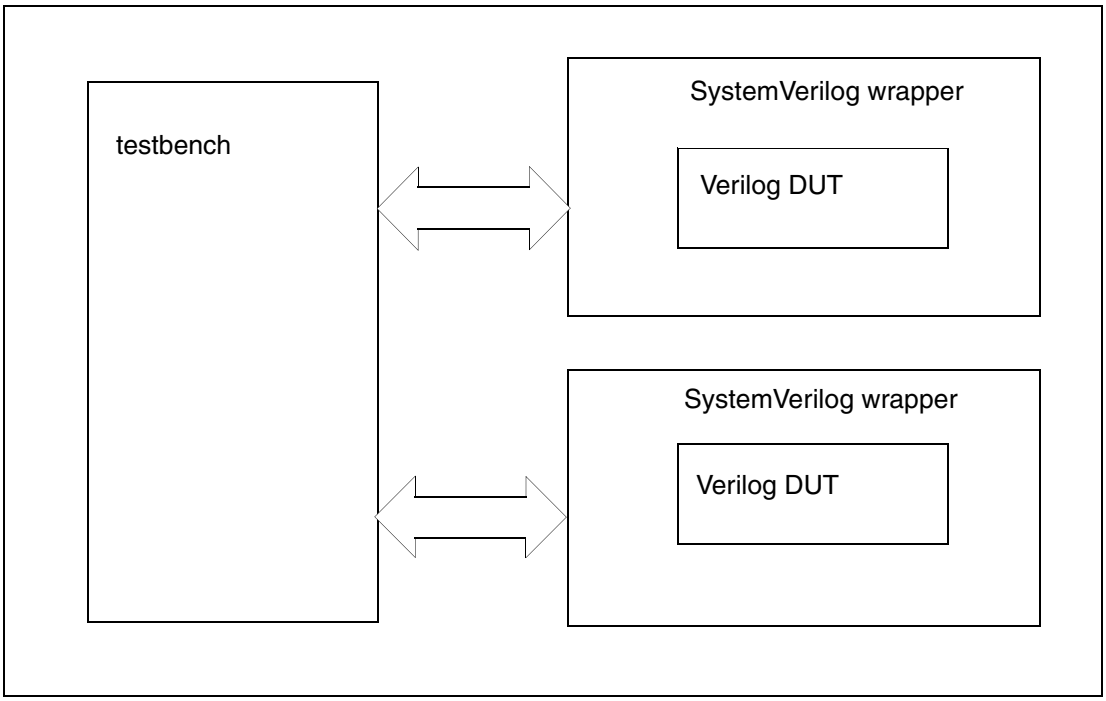


Figure 1-7 Multiple DUT Testing Not Allowed



Elaboration Reports

You can control the type and the amount of information that is included in the elaboration report by setting the `hdlin_reporting_level` variable to `basic`, `comprehensive`, `verbose`, or `none`. Table 1-3 shows what is included in the report based on each setting. In the table, true indicates that the information will be included in the report, false indicates that it will not be included in the report, and verbose indicates that the report will include detailed information. If you do not specify a setting, `hdlin_reporting_level` is set to `basic` by default.

Table 1-3 Basic hdlin_reporting_level Variable Settings

Information included in report	`none`	`basic`	`comprehensive`	`verbose`
`floating_net_to_ground` Reports the floating net to ground connections.	false	false	true	true
`fsm` Prints a report of inferred state variables.	false	false	true	true

Table 1-3 Basic hdlin_reporting_level Variable Settings (Continued)

Information included in report	none	basic	comprehensive	verbose
inferred_modules Prints a report of inferred sequential elements.	false	true	true	verbose
mux_op Prints a report of MUX_OPs.	false	true	true	true
syn_cell Prints a report of synthetic cells.	false	false	true	true
tri_state Prints a report of inferred three-state elements.	false	true	true	true

In addition to the basic settings, you can also specify the add (+) or subtract (-) options to customize a report. For example, if you want a report to include floating-net-to-ground connections, synthetic cells, inferred state variables, and verbose information for inferred sequential elements, but you do not want to include MUX_OPs or inferred three-state elements, you can set the `hdlin_reporting_level` variable to the following setting:

```
set hdlin_reporting_level verbose-mux_op-tri_state
```

As another example, if you set the `hdlin_reporting_level` variable to the following setting,

```
set hdlin_reporting_level basic+floating_net_to_ground+syn_cell+fsm
```

HDL Compiler issues a report that is equivalent to `set hdlin_reporting_level comprehensive`, meaning that the elaboration report will include comprehensive information for all the information listed in Table 1-3.

2

Global Name Space ($unit)

This chapter describes how the Synopsys synthesis tool supports SystemVerilog global declarations in the following sections:

- $unit Usage
- Reading Designs That Use $unit
- $unit Synthesis Restrictions

$unit Usage

The Synopsys synthesis tool supports SystemVerilog global declarations by adding the $unit feature. This is a top-level name space outside any module and is visible to all modules at all hierarchical levels. $unit enables sharing of objects between modules. This reduces the overall RTL code size because globally used objects can reside in one common place.

The following objects are supported in $unit:

- Type definitions
- Enumerated types
- Declarations of localparams
  - The parameter keyword can also be used to declare local parameters in $unit.
- Automatic tasks and automatic functions
- Constant declarations
- Simulation-related constructs such as timeunit, timeprecision, and timescale
- Directives
- Verilog 2001 compiler directives such as `include and `define

To illustrate $unit usage, consider Example 2-1. Here, the code combines objects that include enums, typedefs, parameter declarations, tasks, functions, and structure variables in $unit; these objects are used by the module test.

Example 2-1 $unit Usage

```
typedef enum logic {FALSE, TRUE} my_reg;

localparam a = '1;

typedef struct{
my_reg [a:0] orig;
my_reg [a:0] orig_inverted;
} my_struct;

function automatic my_reg [a:0] invert (my_reg [a:0] value);
return (~value);
endfunction

task automatic check_invert(my_struct struct_in);
   begin
      if(struct_in.orig == struct_in.orig_inverted)
         $display("\n ERROR: Value not inverted\n");
      else
         $display("\n CORRECT: Value is inverted\n");
   end
endtask
```

```
module test(input my_reg [a:0] din, output my_struct dout);

   assign dout.orig = din;
   assign dout.orig_inverted = invert(din);

   always_comb check_invert(dout);

endmodule : test
```

In Example 2-2, the code combines objects such as timeunit, timeprecision, and localparams in $unit; these objects are used in module dff_ns. Note: timeprecision and timeunit are simulation-related constructs. The delay control and intraassignment delays for blocking assignments are ignored.

Example 2-2

```
timeunit 1ns;
timeprecision 1ps;

localparam DELAY =  100;

module dff_ns(input logic clk, d, reset, output logic q);

//synopsys sync_set_reset "reset"
  always@(posedge clk)
    begin
      if(reset) #DELAY q = 0;
      else q = #1ns d;
    end
endmodule
```

Reading Designs That Use $unit

The following sections describe guidelines for writing Tcl scripts to read designs that use the $unit feature:

- $unit Reading Rule 1—Define Objects Before Use
- $unit Reading Rule 2—Read Global Files First
- $unit Reading Rule 3—Read Global Files for Each Analyze

The examples provided in this section use the `analyze` command, but the reading guidelines apply to both the `read` commands and the `analyze` command.

The following four files are used to describe these guidelines:

- global.sv—Contains the following global declaration used by other modules:

  ```
  typedef struct {
  ```

```
logic a, b;} data;
```

- and_struct.sv—Contains the module and_struct, which uses the global declaration. Here, the input is assigned the type data; the type data is defined in $unit.

```
module and_struct(input data din, output a_and_b);
assign a_and_b = din.a & din.b;
endmodule : and_struct
```

- or_struct.sv—Contains the module or_struct, which uses the global declaration. Here, the input is assigned the type data; the type data is defined in $unit.

```
module or_struct(input data din, output a_or_b);
assign a_or_b = din.a | din.b;
endmodule : or_struct
```

- top.sv—contains the module top, which instantiates the and_struct instance as u1 and instantiates the or_struct instance as u2. In design top, the input is assigned the type data; the type data is defined in $unit.

```
module top(input data din, output or_result, and_result);
and_struct u1(.din, .a_and_b(and_result));
or_struct u2(.din, .a_or_b(or_result));
endmodule : top
```

$unit Reading Rule 1—Define Objects Before Use

You must define an object before you use it. Consider Example 2-3.

Example 2-3

```
analyze -f sverilog {global.sv and_struct.sv or_struct.sv top.sv}
elaborate top
write -f verilog -h -o gtech.sample1.v
quit
```

In Example 2-3, one `analyze` command reads all the files and the global.sv file is the first file in the read list. The and_struct.sv, or_struct.sv, and top.sv files, which use the global declaration, are listed after the global.sv file.

Because the global.sv file is read first, the tool generates the netlist without errors as shown in Example 2-4.

Example 2-4

```
dc_shell> analyze -f sverilog {global.sv and_struct.sv or_struct.sv top.sv}
Running PRESTO HDLC
Searching for ./global.sv
Searching for ./and_struct.sv
Searching for ./or_struct.sv
Searching for ./top.sv
Compiling source file ./global.sv
Compiling source file ./and_struct.sv
```

```
Compiling source file ./or_struct.sv
Compiling source file ./top.sv
Presto compilation completed successfully.
Loading db file '/remote/release/synthesis/design_compiler/libraries/syn/lsi_10k.db'
Loading db file '/remote/release/synthesis/design_compiler/libraries/
syn/dw_foundation.sldb'
1
dc_shell> elaborate top
Loading db file '/remote/release/synthesis/design_compiler/libraries/syn/gtech.db'
Loading db file '/remote/release/synthesis/design_compiler/libraries/syn/
standard.sldb'
  Loading link library 'lsi_10k'
  Loading link library 'gtech'
Running PRESTO HDLC
Presto compilation completed successfully.
Elaborated 1 design.
Current design is now 'top'.
Information: Building the design 'and_struct'. (HDL-193)
Presto compilation completed successfully.
Information: Building the design 'or_struct'. (HDL-193)
Presto compilation completed successfully.
1
```

However, if the global file is read after the design files, as shown in Example 2-5, the tool generates the following error message:

```
 Error:  ./and_struct.sv:1: Syntax error at or near token ’din’.

(VER-294)
```

Example 2-5

```
analyze -f sverilog {and_struct.sv or_struct.sv top.sv global.sv}
elaborate top
write -f verilog -h -o gtech.sample2_error.v
quit
```

Example 2-6 shows the log containing the error message.

Example 2-6

```
dc_shell> analyze -f sverilog {and_struct.sv or_struct.sv top.sv global.sv}
Running PRESTO HDLC
Searching for ./and_struct.sv
Searching for ./or_struct.sv
Searching for ./top.sv
Searching for ./global.sv
Compiling source file ./and_struct.sv
Error:  ./and_struct.sv:1: Syntax error at or near token '': missing comma in list of
port declarations. (VER-294)
Error:  ./and_struct.sv:1: Syntax error at or near token 'din'. (VER-294)
Compiling source file ./or_struct.sv
Error:  Cannot recover from previous errors. (VER-518)
*** Presto compilation terminated with 3 errors. ***
0
```

$unit Reading Rule 2—Read Global Files First

When reading files individually, you must always include all applicable global files in the read. Alternatively, you can use the `include construct as shown in Example 2-11 and Example 2-12. However, $unit method is recommended over the `include method because $unit enables easier tracking of common functions, tasks, structures, unions, and typedefs, which can all reside in one common $unit area (outside any module). The tool reports an error message if you mix both `include and $unit.

Example 2-7 illustrates reading files individually.

Example 2-7

```
analyze -f sverilog {global.sv and_struct.sv}
analyze -f sverilog {global.sv or_struct.sv}
analyze -f sverilog {global.sv top.sv}
elaborate top
write -f verilog -h -o gtech.sample3.v
quit
```

In Example 2-7, a separate `analyze` command reads and_struct.sv, or_struct.sv, and top.sv. Because $unit is reset to empty with each command, you must always first include the global file in the read list for each `analyze` command.

Because all required global declarations are available in $unit when the modules are read, the tool generates the netlist without any errors as shown in Example 2-8.

Example 2-8

```
dc_shell> analyze -f sverilog {global.sv and_struct.sv}
Running PRESTO HDLC
Searching for ./global.sv
Searching for ./and_struct.sv
Compiling source file ./global.sv
Compiling source file ./and_struct.sv
Presto compilation completed successfully.
Loading db file '/remote/release/synthesis/design_compiler/libraries/syn/lsi_10k.db'
Loading db file '/remote/release/synthesis/design_compiler/libraries/
syn/dw_foundation.sldb'
1
dc_shell> analyze -f sverilog {global.sv or_struct.sv}
Running PRESTO HDLC
Searching for ./global.sv
Searching for ./or_struct.sv
Compiling source file ./global.sv
Compiling source file ./or_struct.sv
Presto compilation completed successfully.
1
dc_shell> analyze -f sverilog {global.sv top.sv}
Running PRESTO HDLC
Searching for ./global.sv
Searching for ./top.sv
Compiling source file ./global.sv
Compiling source file ./top.sv
```

```
Presto compilation completed successfully.
1
dc_shell> elaborate top
Loading db file '/remote/release/synthesis/design_compiler/libraries/syn/gtech.db'
Loading db file '/remote/release/synthesis/design_compiler/libraries/syn/
standard.sldb'
  Loading link library 'lsi_10k'
  Loading link library 'gtech'
Running PRESTO HDLC
Presto compilation completed successfully.
Elaborated 1 design.
Current design is now 'top'.
Information: Building the design 'and_struct'. (HDL-193)
Presto compilation completed successfully.
Information: Building the design 'or_struct'. (HDL-193)
Presto compilation completed successfully.
1
```

However, if the global file is not read first and with each individual design file, as shown in Example 2-9, the tool reports an error, which is shown in Example 2-10.

Example 2-9

```
analyze -f sverilog {global.sv and_struct.sv}
analyze -f sverilog {or_struct.sv}
analyze -f sverilog {global.sv top.sv}
elaborate top
write -f verilog -h -o gtech.sample4_error.v
quit
```

In Example 2-9, the global.sv file is not read before the or_struct.sv file. In this case, the tool reports the following error because the structure type data is applied to din before it is defined:

```
Error:  ./or_struct.sv:1: Syntax error at or near token 'din'. (VER-294)
```

Example 2-10 shows the log results containing the error.

Example 2-10

```
analyze -f sverilog {global.sv and_struct.sv}
Running PRESTO HDLC
Searching for ./global.sv
Searching for ./and_struct.sv
Compiling source file ./global.sv
Compiling source file ./and_struct.sv
Presto compilation completed successfully.
Loading db file '/remote/release/synthesis/design_compiler/libraries/syn/lsi_10k.db'
Loading db file '/remote/release/synthesis/design_compiler/libraries/
syn/dw_foundation.sldb'
1
dc_shell> analyze -f sverilog {or_struct.sv}
Running PRESTO HDLC
Searching for ./or_struct.sv
Compiling source file ./or_struct.sv
Error:  ./or_struct.sv:1: Syntax error at or near token '': missing comma in list of
port
```

```
declarations. (VER-294)
Error:  ./or_struct.sv:1: Syntax error at or near token 'din'. (VER-294)
*** Presto compilation terminated with 2 errors. ***
0
dc_shell> analyze -f sverilog {global.sv top.sv}
Running PRESTO HDLC
Searching for ./global.sv
Searching for ./top.sv
Compiling source file ./global.sv
Compiling source file ./top.sv
Presto compilation completed successfully.
1
```

Although not recommended for the reasons described earlier, you can the use `include method to fix the problem in Example 2-10. This method is shown in Example 2-11.

Example 2-11

```
`include "global.sv"

module or_struct(input data din, output a_or_b);
      assign a_or_b = din.a | din.b;
endmodule : or_struct
```

Example 2-12 shows the read script for the `include method.

Example 2-12

```
analyze -f sverilog {global.sv and_struct.sv}
analyze -f sverilog {or_struct_modified.sv}
analyze -f sverilog {global.sv top.sv}
elaborate top
write -f verilog -h -o gtech.sample5.v
quit
```

The tool reads all the files and generates the netlist without any errors as shown in Example 2-13.

Example 2-13

```
dc_shell> analyze -f sverilog {global.sv and_struct.sv}
Running PRESTO HDLC
Searching for ./global.sv
Searching for ./and_struct.sv
Compiling source file ./global.sv
Compiling source file ./and_struct.sv
Presto compilation completed successfully.
1
dc_shell> analyze -f sverilog {or_struct_modified.sv}
Running PRESTO HDLC
Searching for ./or_struct_modified.sv
Compiling source file ./or_struct_modified.sv
Opening include file global.sv
Presto compilation completed successfully.
1
dc_shell> analyze -f sverilog {global.sv top.sv}
Running PRESTO HDLC
Searching for ./global.sv
```

```
Searching for ./top.sv
Compiling source file ./global.sv
Compiling source file ./top.sv
Presto compilation completed successfully.
1
dc_shell> elaborate top
Loading db file '/remote/release/synthesis/design_compiler/libraries/syn/gtech.db'
Loading db file '/remote/release/synthesis/design_compiler/libraries/syn/
standard.sldb'
  Loading link library 'lsi_10k'
  Loading link library 'gtech'
Running PRESTO HDLC
Presto compilation completed successfully.
Elaborated 1 design.
Current design is now 'top'.
Information: Building the design 'and_struct'. (HDL-193)
Presto compilation completed successfully.
Information: Building the design 'or_struct'. (HDL-193)
Presto compilation completed successfully.
1
```

$unit Reading Rule 3—Read Global Files for Each Analyze

For each `analyze` command, you must read all global files that are used by the design files analyzed. A separate $unit is stored for each module read, and it incorporates all the $unit objects seen in the current command.

The following five files are used to describe this restriction:

- global.sv - This file contains a global declaration, the structure data type, used by other modules.

  ```
  typedef struct {
  logic a, b;} data;
  ```

- global2.sv - This file contains the global function parity that uses the structure type data from the global.sv file as the input type.

  ```
  function automatic parity(input data din);
  return(^{din.a, din.b});
  endfunction
  ```

- and_struct_exor.sv - This file contains the module and_struct_exor. The design uses the structure type data from the global.sv file as the input type for din; the design computes the parity of din using the parity function from the global2.sv file.

  ```
  module and_struct_exor(input data din, output a_and_b,
  a_exor_b);
  assign a_and_b = din.a & din.b;
  assign a_exor_b = parity(din);
  endmodule : and_struct_exor
  ```

- or_struct.sv - This file contains the module or_struct, which uses the structure type data from the global.sv file as the input type for din.

```
module or_struct(input data din, output a_or_b);
assign a_or_b = din.a | din.b;
endmodule : or_struct
```

- top_modified.sv - This file contains the module top, which instantiates the and_struct_exor instance as u1 and instantiates the or_struct instance as u2. The design uses the structure type data from the global.sv file as the input type for din.

```
module top(input data din, output or_result, and_result,
parity_result);

and_struct_exor u1(.din, .a_and_b(and_result),
.a_exor_b(parity_result));
or_struct u2(.din, .a_or_b(or_result));

endmodule : top
```

To understand this restriction, consider Example 2-14, which reads the complete design into the tool.

Example 2-14

```
analyze -f sverilog {global.sv global2.sv and_struct_exor.sv}
analyze -f sverilog {global.sv or_struct.sv}
analyze -f sverilog {global.sv top_modified.sv}
elaborate top
write -f verilog -h -o gtech.sample6.v
quit
```

In Example 2-14, the first `analyze` command reads the global files before the design files so that the two objects (function parity and the structure data) are available to the module and_struct_exor. When reading and_struct_exor, you must read global.sv and global2.sv first as shown in Example 2-14 and below:

```
analyze -f sverilog {global.sv global2.sv
and_struct_exor.sv}
```

Because $unit is set to empty at each `analyze` command, global.sv must be read again for module or_struct.

The tool generates the netlist without any errors, as shown in Example 2-15, because all required global declarations are available in $unit when the modules are read.

Example 2-15

```
dc_shell> analyze -f sverilog {global.sv global2.sv and_struct_exor.sv}
Running PRESTO HDLC
Searching for ./global.sv
Searching for ./global2.sv
Searching for ./and_struct_exor.sv
Compiling source file ./global.sv
```

```
Compiling source file ./global2.sv
Compiling source file ./and_struct_exor.sv
Presto compilation completed successfully.
Loading db file '/remote/release/synthesis/design_compiler/libraries/syn/lsi_10k.db'
Loading db file '/remote/release/synthesis/design_compiler/libraries/
syn/dw_foundation.sldb'
1
dc_shell> analyze -f sverilog { global.sv or_struct.sv}
Running PRESTO HDLC
Searching for ./global.sv
Searching for ./or_struct.sv
Compiling source file ./global.sv
Compiling source file ./or_struct.sv
Presto compilation completed successfully.
1
dc_shell> analyze -f sverilog {global.sv top_modified.sv}
Running PRESTO HDLC
Searching for ./global.sv
Searching for ./top_modified.sv
Compiling source file ./global.sv
Compiling source file ./top_modified.sv
Presto compilation completed successfully.
1
dc_shell> elaborate top
Loading db file '/remote/release/synthesis/design_compiler/libraries/syn/gtech.db'
Loading db file '/remote/release/synthesis/design_compiler/libraries/syn/
standard.sldb'
  Loading link library 'lsi_10k'
  Loading link library 'gtech'
Running PRESTO HDLC
Presto compilation completed successfully.
Elaborated 1 design.
Current design is now 'top'.
Information: Building the design 'and_struct_exor'. (HDL-193)
Presto compilation completed successfully.
Information: Building the design 'or_struct'. (HDL-193)
Presto compilation completed successfully.
1
```

$unit Synthesis Restrictions

The following sections describe synthesis restrictions for $unit:

- $unit Restriction 1—Declarations
- $unit Restriction 2—Instantiations
- $unit Restriction 3—Static Variables
- $unit Restriction 4—Static Tasks and Functions

$unit Restriction 1—Declarations

Declarations of nets and variables in $unit are not allowed. For example, the following in $unit is not synthesizable:

```
logic a, c;
wire b;
```

$unit Restriction 2—Instantiations

Module, interface, and gate instantiations in $unit are not allowed.

For example, the following code in $unit is not synthesizable:

```
and and_gate(out, in1, in2);
half_adder U1(sum, ain, bin); // where half_adder is module
nameiface if1();   // where iface is the interface name
```

$unit Restriction 3—Static Variables

Static variables inside automatic functions or automatic tasks in $unit are not supported.

For example, the following code is not synthesizable in $unit:

```
function automatic [31:0] incr_by_value(logic [31:0] val);
static logic [31:0] sum = 0; //static variable here
sum += val;
return(sum);
endfunction
```

$unit Restriction 4—Static Tasks and Functions

Static tasks or static functions in $unit are not supported for synthesis. For example, the following code is not supported:

```
function static logic non_zero_int_is_true (logic [31:0] val);
if (val == 0) return ('0);
else return('1);
endfunction
```

In these cases, the tool returns an error message similar to the following:

```
Error:  ...: Static function 'adder' is not synthesizable
in $unit, expecting "automatic" keyword (VER-523)
```

Verilog functions are static by default, so unless you use the keyword automatic, the tool assumes a static function and returns a VER-523 error message. This code

```
function void adder (input cin, [7:0] in1, [7:0] in2,
                     output [8:0] result);
  begin
    assign result[0] = cin;
  end
endfunction
```

returns a VER-523 error message.

3

Packages

In SystemVerilog, you can use packages as a mechanism for sharing parameters, types, tasks, and functions among multiple SystemVerilog modules and interfaces. Packages are explicitly named scopes appearing at the same level as top-level modules. Parameters, types, tasks, and functions can be declared within a package. The following sections in this chapter describe the different ways you can reference such declarations within modules, interfaces, and other packages:

Benefits of Using Packages

In any SystemVerilog design project, it is common to have types, functions, and tasks that are used repetitively by the design team. When you put these common constructs inside packages, they can be shared among the design team. This allows developers to use existing code based on their needs and requirements without any ambiguity.

After all the types, functions, and tasks are finalized, the package file is analyzed. Modules that use the package information can be analyzed separately without the need to re-analyze the package again. This can save runtime for large packages.

Recommended Use Model

To read in packages,

1. Analyze your package. This creates a `package_name.pvk` temporary file. If you modify your package by adding or removing functions, tasks, types, and so on, the tool overwrites your temporary file and issues a VER-26 warning to indicate this change, as shown below.

```
Warning:  ./test.sv:1: The package p has already been analyzed. It is
being replaced. (VER-26)
```

2. Analyze and elaborate the modules that use the packages.

Note:
If your modules were analyzed using previous versions of the package, re-analyze and elaborate the modules again so that the tool picks up the most up-to-date declarations from your packages.

The next section shows a sample script.

Example 1: Package and Scope Extraction in Multiple Modules

In this example, the package.sv file contains the package `pkg1` that holds the types `my_T` and `my_struct`. It also contains the functions `subtract` and `complex_add`.

The scope extraction operator `::` is used to extract the function and types using the syntax `package_name::type_name` or `package_name::function_name`. The `my_T` type and `subtract` function are used by module `test1` to compute the `result` and `equal` values. The `my_struct` type and `complex_add` function are used by the second module, `test2`, to compute `result2`. When you analyze the package file first, the tool creates the temporary file pkg1.pvk. Later, the modules could be analyzed and elaborated by themselves because the pkg1.pvk file exists.

Example 3-1 RTL Code File1: package.sv

```
package pkg1;
typedef struct {int a;logic b;} my_struct;
typedef logic [127:0] my_T;

function automatic my_T subtract(my_T one, two);
return(one - two);
endfunction

function automatic my_struct complex_add(my_struct one, two);
complex_add.a = one.a + two.a;
complex_add.b = one.b + two.b;
endfunction

endpackage : pkg1
```

Example 3-2 File2: test1.sv

```
module test1(input pkg1::my_T in1, in2, output  pkg1::my_T result, input
[127:0]test_vector, output equal);

assign result = pkg1::subtract(in1, in2);
assign equal = (in1 == test_vector);

endmodule
```

Example 3-3 File3: test2.sv

```
module test2(input pkg1::my_struct in1, in2, output  pkg1::my_struct
result2);

assign result2 = pkg1::complex_add(in1, in2);

endmodule
```

Example 3-4 Scripts

```
# Analyze the package file the first time, it will creates pkg1.pvk file
:
analyze -f sverilog package.sv

# Analyze/elaborate the first module, test1, that uses the package
information:
analyze -f sverilog test1.sv
elaborate test1

# Analyze/elaborate the second module, test2, that uses the package
information:
analyze -f sverilog test2.sv
elaborate test2
```

Note:

You could also analyze all the packages and modules at the same time.

The examples in the next two sections discuss two alternative sharing techniques called `wildcard` imports and `specific` imports.

Example 2: Wildcard Imports From a Package Into a Module

Example 3-5 uses wildcard imports to import both the enum identifier color and its literal values into the module scope. Both imported items are used in a finite state machine.

Example 3-5 Wildcard Imports

```
package p;
typedef enum logic [1:0] {red, blue, yellow, green} color ;
endpackage

module fsm_controller (output logic [2:0] result, input [1:0] read_value, input clock,
  reset) ;

import p::*; // By doing wildcard imports, both the enum identifier and
             // literals become available and are kept, since they
             // are used inside the module.
color State;

always_ff @(posedge clock, negedge reset) begin
  if (!reset) begin
    State = red;
  end
  else begin
    State = color'(read_value);
  end
end

always_comb begin
  case(State)
    red    : result = 3'b101;
    yellow : result = 3'b001;
    blue   : result = 3'b000;
    green  : result = 3'b010;
  endcase
end

endmodule
```

Example 3: Specific Imports From a Package Into a Module

Packages can hold many declarations. However, all of the declarations might not be required by a certain module. In such cases, you can import the specific declarations that you need for your module. This is quite common if the entire design team uses a single package for all of their declarations. In these cases, you can import your specific types or functions from the global package.

Consider a package p that contains the following types: color, packet_t, SWITCH_VALUES, and sw_lgt_pair:

Example 3-6 Package p

```
package p;
typedef enum logic [1:0] {red, blue, yellow, green} color ;
typedef struct {logic [7:0] src; logic [7:0] dst; logic [31:0] data;} packet_t;
typedef enum logic {OFF, ON} SWITCH_VALUES;

typedef struct packed {
  SWITCH_VALUES switch; // 1 bit
  color light;          // 2 bit
  logic test_bit;       // 1 bit
} sw_lgt_pair;          //4 bits
endpackage
```

If the fsm_controller module uses only color, then it could be coded using specific imports as follows:

```
module fsm_controller (output logic [2:0] result, input [1:0] read_value,
input
   clock,reset) ;
import p::color; //use specific imports to import the identifier color
import p::red;   //use specific imports to import the enum literal values
import p::blue;
import p::yellow;
import p::green;

color State;

always_ff @(posedge clock, negedge reset) begin
  if (!reset) begin
    State = red;
  end
  else begin
    State = color'(read_value);
  end
end

always_comb begin
  case(State)
    red    : result = 3'b101;
    yellow : result = 3'b001;
    blue   : result = 3'b000;
    green  : result = 3'b010;
  endcase
end
endmodule
```

Note:
Even if you did wildcard imports by using the `import p::*;` sharing technique, the module only uses what is actually kept. When creating large designs with packages, use `specific` imports so that you know what types were imported into your design. If you use `wildcard` imports, debugging might be harder because you have to step through your entire module to figure out what was actually used.

Wildcard Import into $unit

The tool supports wildcard imports into $unit from packages, facilitating flexible ports in module header declarations. Example 3-7 shows the code compaction benefit of wildcard usage.

Example 3-7

```
package pkg;
 typedef struct {byte a, b;} packet;
 typedef enum logic[1:0] {ONE,TWO,THREE} state_t;
endpackage:pkg

import pkg::*; //wildcard import into $unit

module test (output packet packet1, input clk, rst, input state_t data1,
data2);
// Scope extraction style: module test (output pkg::packet packet1, input
clk, rst,
input
// pkg::state_t data1, data2); without imports in $unit
. . .
endmodule
```

Restrictions

Use the following package usage guidelines:

- Wire and variable declarations in packages are not supported. The tool reports an error.
- Functions and tasks need to be automatic when declared inside packages.
- Sequence, property, and program blocks are ignored. The tool reports a warning. For more information, see “Reading Assertions in Synthesis” in Chapter 9.

4

Inferring Combinational Logic

This chapter provides coding examples for inferring combinational logic by using the SystemVerilog constructs always_comb, unique if, priority if, and priority case. Additionally, the new SystemVerilog operators, such as +=, -=, *=, /=, %=, &=, |=, ^= , <<=, >>=, <<<=, and >>>=, are used in an ALU example. For a complete FIFO design example that uses various SystemVerilog constructs, see Appendix A, “SystemVerilog Design Examples.”

This chapter contains the following sections:

always_comb

Example 4-1 uses the always_comb construct to build a priority encoder.

Example 4-1 Inferred Priority Encoder Using always_comb

```
module priority_low_high #(parameter N = 8, parameter log2N = 3 )
                          (input [N-1:0] A, output logic [log2N-1:0] P, logic F);
   function [log2N:0] calc_priority([N-1:0] A);
     logic F;
       begin
          F = 1'b0;
          calc_priority = {3'b0, F};
            for (int I=0; I<N; I++)
                if (A[I])
                    begin
                       F = 1'b1;
                       calc_priority = {I, F}; // Override previous index
                     end
        end
    endfunction

        always_comb
           begin
             {P, F} = calc_priority(A);
           end
endmodule
```

Using always_comb and Inferring a Register

Even though you use the always_comb construct, if your code requires a holding state, such as in Example 4-2, the tool will build a latch and generate the warning shown in Example 4-3. Example 4-4 shows the resulting GTECH netlist and Example 4-5 shows the resulting gate-level netlist.

When the always_ff, always_latch, and always_comb constructs are used and the intended hardware is not inferred by the tool, the tool generates a warning.

Example 4-2 Latch Built Even Though always_comb Used

```
module top(input logic a, b, output logic c);
always_comb
  case(a)
    1'b1 : c = b;
  endcase
endmodule
```

Example 4-3 Warning Reported When a Latch is Built Within an always_comb Block

```
Statistics for case statements in always block at line 2 in file
        './always_comb.inferring.reg.sv'
===============================================
|           Line           |  full/ parallel  |
===============================================
|            3             |     no/auto      |
===============================================

Inferred memory devices in process
        in routine top line 2 in file
                './always_comb.inferring.reg.sv'.
=========================================================================
|    Register Name    | Type  | Width | Bus | MB | AR | AS | SR | SS | ST |
=========================================================================
|        c_reg        | Latch |   1   |  N  | N  | N  | N  | -  | -  | -  |
=========================================================================
Warning:  ./always_comb.inferring.reg.sv:2: Netlist for always_comb block contains a
  latch. (ELAB-974)
Presto compilation completed successfully.
```

Example 4-4 GTECH Netlist

```
module top ( a, b, c );
  input a, b;
  output c;

  \**SEQGEN**  c_reg ( .clear(1'b0), .preset(1'b0), .next_state(1'b0),
        .clocked_on(1'b0), .data_in(b), .enable(a), .Q(c), .synch_clear(1'b0),
        .synch_preset(1'b0), .synch_toggle(1'b0), .synch_enable(1'b0) );
endmodule
```

Example 4-5 Gate-Level Netlist

```
module top ( a, b, c );
  input a, b;
  output c;

  LD1 c_reg ( .G(a), .D(b), .Q(c) );
endmodule
```

Using always_comb with Empty Blocks

When you use the always_comb construct, you tell the compiler that your intended hardware is combinational logic. If your code might not result in this logic, the tool generates a warning message. In Example 4-6, tmp is not in an output path and might be removed during compile. The tool reports the following:

```
Warning:  /../../empty_always.sv:6: Netlist for always_comb
block is empty. (ELAB-982)
```

Example 4-6

```
module test(input a, b, output c);
   logic tmp;
   assign  c = a | b;
   always_comb   tmp = a & b;
endmodule
```

unique if

Example 4-7 shows how to infer multiplexing logic by using the unique if statement.

Example 4-7

```
module unique_if (input a, b, c, d, [3:0] sel, output logic z);
always_comb
begin
unique if (sel[3]) z = d;
  else if (sel[2]) z = c;
  else if (sel[1]) z = b;
  else if (sel[0]) z = a;
end
endmodule
```

priority if

Example 4-8 shows how to infer multiplexing logic by using the priority if construct.

Example 4-8

```
module priority_if (input a, b, c, d, [3:0] sel, output logic z);
always_comb
begin
priority if (sel[3]) z = d;
    else if (sel[2]) z = c;
    else if (sel[1]) z = b;
    else if (sel[0]) z = a;
end
endmodule
```

priority case

Example 4-9 shows how to create combinational logic by using the priority case construct. This code infers multiplexing logic. Coding with the priority keyword on a case statement with no default is the same as using the Synopsys `full_case` directive; however, the keyword is recommended to prevent simulation/synthesis mismatch, which can occur when the directive is used. For additional information, see “Reducing Case Mismatches” on page 9-7.

Example 4-9

```
module my_priority_case (input [1:0] in, a, b, c,
                         output logic [1:0] out);

  always_comb
    priority case (in)
       0: out = a;
       1: out = b;
       2: out = c;
    endcase

endmodule
```

unique case

Example 4-10 builds a state machine and uses the unique case construct for the state control. Coding with the unique keyword on a case statement is the same as using the Synopsys `full_case` and `parallel_case` directives; however, the keyword is recommended to prevent simulation/synthesis mismatch, which can occur when the directives are used. For additional information, see “Reducing Case Mismatches” on page 9-7.

Example 4-10

```
module fsm_cc1_3oh (input in1,a,b,c,d,clk, output logic o1);

logic [3:0] state, next;

always_ff @(posedge clk)
 state <= next;

always_comb
  begin
    next = state;
    unique case (1'b1)
    state[0] : begin
                  next[0] = (in1 == 1'b1); o1 = a;
               end
    state[1] : begin
                  next[1] = 1’b1; o1 = b;
               end
    state[2] : begin
                  next[2] = 1'b1; o1 = c;
               end
    state[3] : begin
                  next[3] = 1'b1; o1 = d;
               end
    endcase
```

```
  end

endmodule
```

New Operators

Example 4-11 builds an ALU by using the new SystemVerilog operators: +=, -=, *=, /=, %=, &=, |=, ^= , <<=, >>=, <<<=, and >>>=.

Example 4-11

```
module ALU (input logic clk, reset_n, alu_a_in, alu_b_in, alu_output_en,
load_inputs, logic [3:0] control_in, output logic [15:0] alu_result);
logic [15:0] alu_temp;
logic [7:0] alu_a, alu_b;
logic [3:0] control;
/*****************************************************************
* Registered inputs to ALU
*****************************************************************/
always_ff @ (posedge clk, negedge reset_n)
if (!reset_n)
begin
  alu_a <= 0;
  alu_b <= 0;
  control <= 0;
end
else if(load_inputs)
begin
  alu_a <= alu_a_in;
  alu_b <= alu_b_in;
  control <= control_in;
end
/*****************************************************************
* ALU execution based on control signal
*****************************************************************/
always_comb
begin
  alu_temp = alu_a;
  case (control)
    4'h0 : alu_temp = alu_a;
    4'h1 : alu_temp = alu_b;
    4'h2 : alu_temp += alu_b;
    4'h3 : alu_temp -= alu_b;
    4'h4 : alu_temp *= alu_b;
    4'h5 : alu_temp /= alu_b;
    4'h6 : alu_temp %= alu_b;
    4'h7 : alu_temp &= alu_b;
    4'h8 : alu_temp |= alu_b;
    4'h9 : alu_temp ^= alu_b;
    4'ha : alu_temp <<= alu_b;
    4'hb : alu_temp >>= alu_b;
    4'hc : alu_temp <<<= alu_b;
    4'hd : alu_temp >>>= alu_b;
    4'he : alu_temp = signed'(alu_a) + signed'(alu_b);
    4'hf : alu_temp = unsigned'(alu_a) + unsigned'(alu_b);
```

```
  endcase
end
/*******************************************************
* Load the ALU result to output
*******************************************************/
assign alu_result = alu_output_en ? 'z : alu_temp;
endmodule : ALU
```

5

Inferring Sequential Logic

This chapter provides coding examples for inferring latches, flip-flops, and master-slave latches using the SystemVerilog constructs always_ff and always_ latch. For a complete FIFO design example that uses the always_ff construct, see Appendix A, “SystemVerilog Design Examples.”

This chapter contains the following sections:

Inferring Latches Using always_latch

This section provides the following examples for inferring various types of latches:

- Example 5-1, “Basic D Latch Using always_latch”
- Example 5-2, “D Latch With Asynchronous Reset Using always_latch”
- Example 5-3, “D Latch With Asynchronous Set Using always_latch”
- Example 5-4, “D Latch With Asynchronous Set and Reset Using always_latch”
- Example 5-5, “SR Latch With Asynchronous Reset Using always_latch”

Example 5-1 shows how to infer a basic D latch using the always_latch construct.

Example 5-1 Basic D Latch Using always_latch

```
module d_latch_alat (input GATE, DATA, output logic Q);
always_latch
  if (GATE) Q = DATA;
endmodule
```

Example 5-2 shows how to infer a D latch with an asynchronous reset using the always_latch construct.

Example 5-2 D Latch With Asynchronous Reset Using always_latch

```
module d_latch_async_reset_alat (input RESET, GATE, DATA, output logic Q);

   //synopsys async_set_reset "RESET"
   always_latch
      if (~RESET)
       Q = 1'b0;
      else if (GATE)
       Q = DATA;

endmodule
```

Example 5-3 shows how to infer a D latch with an asynchronous set using the always_latch construct.

Example 5-3 D Latch With Asynchronous Set Using always_latch

```
module d_latch_async_set_alat (input GATE, DATA, SET, output logic Q);

         //synopsys async_set_reset "SET"
         always_latch
               if (~SET)
                Q = 1'b1;
               else if (GATE)
                Q = DATA;

endmodule
```

Example 5-4 shows how to infer a D latch with an asynchronous set and reset using the always_latch construct.

Example 5-4 D Latch With Asynchronous Set and Reset Using always_latch

```
module d_latch_async_alat (input GATE, DATA, RESET, SET, output logic Q);
// synopsys async_set_reset "RESET, SET"
always_latch
begin : infer
  if (~SET)
    Q = 1'b1;
  else if (~RESET)
    Q = 1'b0;
  else if (GATE)
    Q = DATA;
end
endmodule
```

Example 5-5 shows how to infer an SR latch with an asynchronous reset using the always_latch construct.

Example 5-5 SR Latch With Asynchronous Reset Using always_latch

```
module sr_latch_alat (input SET, RESET, output logic Q);

   //synopsys async_set_reset "SET,RESET"
   always_latch
      if (~RESET)
       Q = 0;
      else if (~SET)
       Q = 1;

endmodule
```

Using always_latch and Not Inferring a Sequential Device

Even though you use the always_latch construct, the tool might not build a sequential device because of your coding style. In these cases, the tool will generate the warning shown in Example 5-6.

Example 5-6 Warning Message

```
Warning:
...../warn_always_latch.sv:3:
Behavior within always_latch block does not represent latched logic.
(ELAB-975)
```

When the always_ff, always_latch, and always_comb constructs are used and the intended hardware is not inferred by the tool, the tool generates a warning.

Using always_latch With Empty Blocks

When you use the always_latch construct, you tell the compiler that your intended hardware is a level-sensitive register. If your code might not result in this logic, the tool generates a warning message. In Example 5-7, `tmp` is not in an output path and might be removed during compile. The tool reports an ELAB-983 warning.

Example 5-7

```
module empty_always_latch(input logic clk, in, output logic out);
  logic tmp;
  always_latch begin
    if(clk)
      tmp <= in;
    end
endmodule
```

Inferring Flip-Flops Using always_ff

This section provides the following examples for inferring various types of flip-flops:

- Example 5-8, Basic D Flip-Flop
- Example 5-9, D Flip-Flop With Synchronous Set Using always_ff
- Example 5-10, D Flip-Flop With Synchronous Reset Using always_ff

Example 5-8 shows how to infer a basic D flip-flop using the always_ff construct.

Example 5-8 Basic D Flip-Flop

```
module dff (input DATA, CLK, output logic Q);
   always_ff @(posedge CLK)
   Q <= DATA;
endmodule
```

Example 5-9 shows how to infer a D flip-flop with a synchronous set using the always_ff construct.

Example 5-9 D Flip-Flop With Synchronous Set Using always_ff

```
module dff_sync_set_aff (input DATA, CLK, SET, output logic Q);

  //synopsys sync_set_reset "SET"
  always_ff @(posedge CLK)
  if (SET)
     Q <= 1'b1;
  else
     Q <= DATA;
endmodule
```

The tool then issues the following memory inference report:

```
Inferred memory devices in process
        in routine dff_sync_set_aff line 4 in file
                '.../eg_4_9.sv'.
===============================================================================
|    Register Name    |   Type    | Width | Bus | MB | AR | AS | SR | SS | ST |
===============================================================================
|        Q_reg        | Flip-flop |   1   |  N  | N  | N  | N  | N  | Y  | N  |
===============================================================================
```

Example 5-10 shows how to infer a D flip-flop with a synchronous reset using the always_ff construct.

Example 5-10 D Flip-Flop With Synchronous Reset Using always_ff

```
module dff_sync_reset_aff (input DATA, CLK, RESET, output logic Q);

         //synopsys sync_set_reset "RESET"
         always_ff @(posedge CLK)
               if (~RESET)
                Q <= 1'b0;
               else
                Q <= DATA;

endmodule
```

Using always_ff and Not Inferring a Sequential Device

Even though you use the always_ff construct, the tool might not build a sequential device because of your coding style. In these cases, the tool generates the warning shown in Example 5-11.

Example 5-11 Warning Message

```
Warning:  ...../warn_always_ff.sv:4:
Behavior within always_ff block does not represent sequential logic.
(ELAB-976)
```

When the always_ff, always_latch, and always_comb constructs are used and the intended hardware is not inferred by the tool, the tool generates a warning.

Using always_ff With Empty Blocks

When you use the always_ff construct, you tell the compiler that your intended hardware is an edge-sensitive register. If your code might not result in this logic, the tool generates a warning message. In Example 5-7, tmp is not in an output path and might be removed during compile. The tool reports an ELAB-984 warning.

Example 5-12

```
module empty_alway_ff (input logic clk, in, output logic out);
  logic tmp;
  always_ff @(posedge clk) begin
    tmp <= in;
  end
endmodule
```

Inferring Master-Slave Latches Using always_ff

You describe the master-slave latch as a flip-flop by using only the slave clock. Specify the master clock as an input port, but do not connect it. In addition, attach the `clocked_on_also` attribute to the master clock port (called MCK in these examples).

This coding style requires that cells in the target technology library have slave clocks defined in the library with the `clocked_on_also` attribute in the cell's state declaration. (For more information, see the Synopsys Library Compiler documentation.)

If Design Compiler does not find any master-slave latches in the target technology library, the tool leaves the master-slave generic cell (MSGEN) unmapped. Design Compiler does not use D flip-flops to implement the equivalent functionality of the cell.

Note:

Although the vendor's component behaves as a master-slave latch, Library Compiler supports only the description of a master-slave flip-flop.

This section provides the following examples for inferring master-slave latches:

- Example 5-13, "Master-Slave Latch Using always_ff"
- Example 5-14, "Two Master-Slave Latches Using always_ff"

Example 5-13 shows how to infer a single master-slave latch using the always_ff construct.

Example 5-13 Master-Slave Latch Using always_ff

```
module mslatch_alat (input SCK, MCK, DATA, output logic Q);
// synopsys dc_script_begin
// set_attribute -type string MCK signal_type clocked_on_also
// set_attribute -type boolean MCK level_sensitive true
// synopsys dc_script_end
```

```
always_ff @ (posedge SCK)
  Q <= DATA;
endmodule
```

Example 5-14 shows how to infer two master-slave latches using the always_ff construct.

Example 5-14 Two Master-Slave Latches Using always_ff

```
module mslatch2_alat (input SCK1, SCK2, MCK1, MCK2, D1, D2, output logic Q1,Q2);
// synopsys dc_tcl_script_begin
// set_attribute -type string MCK1 signal_type clocked_on_also
// set_attribute -type boolean MCK1 level_sensitive true
// set_attribute -type string MCK1 associated_clock SCK1
// set_attribute -type string MCK2 signal_type clocked_on_also
// set_attribute -type boolean MCK2 level_sensitive true
// set_attribute -type string MCK2 associated_clock SCK2
// synopsys dc_tcl_script_end
always_ff @ (posedge SCK1)
  Q1 <= D1;
always_ff @ (posedge SCK2)
  Q2 <= D2;
endmodule
```

6

State Machines

SystemVerilog enables you to define integral named constants to variables by using enumerations. This feature enables you to explicitly identify the state assignments to the synthesis tool.

This chapter provides state machine examples that use enumerations in the following sections:

FSM State Diagram and State Table

The first two state machine coding examples in this chapter

- Example 6-1
- Example 6-4

use the state diagram and state table shown in Figure 6-1 and Table 6-1 respectively.

Figure 6-1 Finite State Machine State Diagram

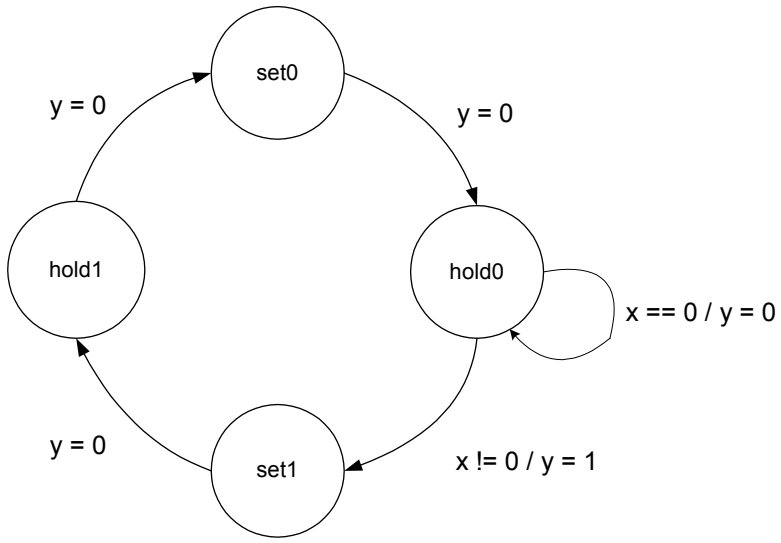


Table 6-1 shows the state table for the state diagram shown in Figure 6-1.

Table 6-1 Finite State Machine State Table

Current state	Input (x)	Next state	Output (Y)
0001 (set0)	0	0010 (hold0)	0
0001 (set0)	1	0010 (hold0)	0
0010 (hold0)	0	0010 (hold0)	0
0010 (hold0)	1	0100 (set1)	1
0100 (set1)	0	1000 (hold1)	0
0100 (set1)	1	1000 (hold1)	0

Table 6-1 Finite State Machine State Table (Continued)

Current state	Input (x)	Next state	Output (Y)
1000 (hold1)	0	0001 (set0)	0
1000 (hold1)	1	0001 (set0)	0

FSM: Using Enumerations Without an Explicit Base Value

To understand how to use enumerations to build a state machine consider Example 6-1. Here, the state assignments to next_state and current_state do not have an explicit base type. Therefore, the base type defaults to int type, which is 32 bits.

Example 6-1

```
module fsm1cs3 (input x, clk, rst, output y);

enum {set0 = 1, hold0 = 2, set1 = 4, hold1 = 8}  current_state, next_state;

  always_ff @ (posedge clk or posedge rst)
    if (rst)
        current_state <= set0;
    else
        current_state <= next_state;

  always_comb
    case (current_state)
        set0:
            next_state = hold0;
        hold0:
            if (x == 0)
                next_state = hold0;
            else
                next_state = set1;
        set1:
            next_state = hold1;
        hold1:
            next_state = set0;
        default :
            next_state = set0;
    endcase
  assign y = current_state == hold0 & x;
endmodule
```

To enable the tool to report state machine inference, set the `hdlin_reporting_level` variable to `comprehensive`, as shown in the script in Example 6-2. The default is `basic`, meaning that an FSM inference report is not generated.

Example 6-2 Script

```
set hdlin_reporting_level basic+fsm
read_sverilog example_6-1.sv
write -f verilog -h -o gtech.example_6-1.v
write -f verilog -h -o gates.example_6-1.v
quit
```

The inference report for the FSM in Example 6-1 is shown in Example 6-3. From this report, you can observe that the FSM inference has 32 bits.

Example 6-3 Inference Report

```
statistics for FSM inference:
  state register: current_state
  states
  ======
  set0:00000000000000000000000000000001
  hold0:00000000000000000000000000000010
  set1:00000000000000000000000000000100
  hold1:00000000000000000000000000001000

  total number of states: 4
```

For more information about the `hdlin_reporting_level` variable, see "Elaboration Reports" on page 1-37.

FSM: Using Enumerations With an Explicit Base Value

In Example 6-1, the FSM code does not specify an explicit base type, which results in an unnecessary 32-bit state encoding. An alternative and preferred way to code this design is shown in Example 6-4. Here, the FSM base value is explicitly set to 4 bits of logic type with the following line of code:

```
enum logic [3:0] {set0 = 4'b0001, hold0 = 4'b0010, set1 =
4'b0100, hold1 = 4'b1000} current_state, next_state;
```

Example 6-4

```
module fsm1cs1 (input logic x, clk, rst, output logic y);

enum logic [3:0] {set0 = 4'b0001, hold0 = 4'b0010, set1 = 4'b0100, hold1 =
4'b1000} current_state, next_state;

   always_ff @ (posedge clk or posedge rst)
    if (rst)
          current_state <= set0;
    else
        current_state <= next_state;
   always_comb
    priority case (current_state)
    set0:
        begin
          next_state = hold0;
          y = 0;
```

```
        end
    hold0:
        begin
          if (x == 0)
              begin
              next_state = hold0;
             y = 0;
              end
      else
         begin
            next_state = set1;
            y = 1;
         end
     end
    set1:
          begin
          next_state = hold1;
          y = 0;
          end
    hold1:
        begin
        next_state = set0;
        y = 0;
        end
    endcase
endmodule
```

Example 6-5 shows the FSM inference report for the state machine code in Example 6-4. From this report you see that the tool uses 4-bit state encoding.

Example 6-5 FSM Inference Report

```
statistics for FSM inference:
  state register: current_state
  states
  ======
  set0:0001
  hold0:0010
  set1:0100
  hold1:1000

  total number of states: 4
```

Automatically Defining the State Variable Values

You do not need to specify all the integral constants for the variables used with enumerations. Consider Example 6-6. Here, the state_assignments are defined as

```
typedef enum logic [3:0] {IDLE = 1, FIVE = 4, TEN, TWENTY_FIVE, FIFTEEN,
THIRTY,
TWENTY, OWE_DIME} state_assignments;

state_assignments D, Q;
```

Notice that only the IDLE state and the FIVE state are explicitly defined while the remaining states—TEN, TWENTY_FIVE, FIFTEEN, THIRTY, TWENTY, OWE_DIME—are not explicitly defined. The tool assigns incremental values of 5, 6, 7, 8, 9, 10, respectively, for these states. This assignment is based on the last explicitly defined state's integral value, which was FIVE = 4.

Example 6-6

```
`define vend_a_drink  {D, dispense, collect} = {IDLE,2'b11}

module drink_machine(input logic reset, clk, nickel_in, dime_in, quarter_in,
                     output logic collect, nickel_out, dime_out, dispense);
typedef enum logic [3:0]{IDLE = 1, FIVE = 4, TEN, TWENTY_FIVE, FIFTEEN,
        THIRTY, TWENTY,OWE_DIME} state_assignments;
state_assignments D, Q;

always_comb
begin
  nickel_out = 0;
  dime_out   = 0;
  dispense   = 0;
  collect    = 0;
  if ( reset ) D = IDLE;
  else begin
    case ( Q )
    IDLE:
      if (nickel_in)       D = FIVE;
      else if (dime_in)    D = TEN;
      else if (quarter_in) D = TWENTY_FIVE;
    FIVE:
      if(nickel_in)        D = TEN;
      else if (dime_in)    D = FIFTEEN;
      else if (quarter_in) D = THIRTY;
    TEN:
      if (nickel_in)       D = FIFTEEN;
      else if (dime_in)    D = TWENTY;
      else if (quarter_in) `vend_a_drink;
    TWENTY_FIVE:
      if( nickel_in)       D = THIRTY;
      else if (dime_in)    `vend_a_drink;
      else if (quarter_in) begin
        `vend_a_drink;
        nickel_out = 1;
```

```
        dime_out = 1;
      end
    FIFTEEN:
      if (nickel_in)       D = TWENTY;
      else if (dime_in)    D = TWENTY_FIVE;
      else if (quarter_in) begin
        `vend_a_drink;
        nickel_out = 1;
      end
    THIRTY:
      if (nickel_in)    `vend_a_drink;
      else if (dime_in) begin
        `vend_a_drink;
        nickel_out = 1;
      end
      else if (quarter_in) begin
        `vend_a_drink;
        dime_out = 1;
        D = OWE_DIME;
      end
    TWENTY:
      if (nickel_in)       D = TWENTY_FIVE;
      else if (dime_in)    D = THIRTY;
      else if (quarter_in) begin
        `vend_a_drink;
        dime_out = 1;
      end
    OWE_DIME:
      begin
        dime_out = 1;
        D = IDLE;
      end
    endcase
  end
end
always_ff@(posedge clk ) begin
  Q <= D;
end
endmodule
```

Example 6-7 shows the FSM inference report.

Example 6-7 FSM Inference Report

```
statistics for FSM inference:
  state register: Q
  states
  ======
  IDLE:  0001
  FIVE:  0100
  TEN:   0101
  TWENTY_FIVE:0110
  FIFTEEN:  0111
  THIRTY:  1000
  TWENTY:  1001
  OWE_DIME:  1010

  total number of states: 8
```

Specify enum Ranges

To simplify state machine coding, the tool allows ranges for enum labels. From the range information, the tool will automatically do the text substitution. Example 6-8 shows the coding style where you must specify each state; Example 6-9 shows the coding style where you just specify the range, and the tool determines each individual state. In the case of Example 6-9, the tool evaluates the state[2] range and determines the individual states to be state0 and state1.

Example 6-8 Specify Each State

```
enum { state0, state1 } current_state;
```

Example 6-9 Specify Enum Range

```
enum { state[2] } current_state;
```

Methods for Enumerated Types

HDL Compiler (Presto Verilog) supports different methods, such as .first, .last, .next, .prev and .num, for accessing different enumerated literals of enumerated types. These methods provide alternative coding techniques for Finite State Machines. You can refer to the LRM for examples.

The following methods are not supported:

- Enum methods for explicit state encoding

  For example, `enum {red = 2'd1, blue = 2'd2, green = 2'd4} state;` does not support enum methods.

- Optional arguments, such as `light.next(5)`

7

Interfaces

By using interface constructs, you can encapsulate intermodule communication, thus making bus specification and bus management easier. Basically, interfaces are a bundle of nets or variables or a combination of both. Additional power can be provided to the interface by embedding tasks, functions, and always blocks inside the interface. In this way, they can act as communication protocol handlers in addition to managing connectivity.

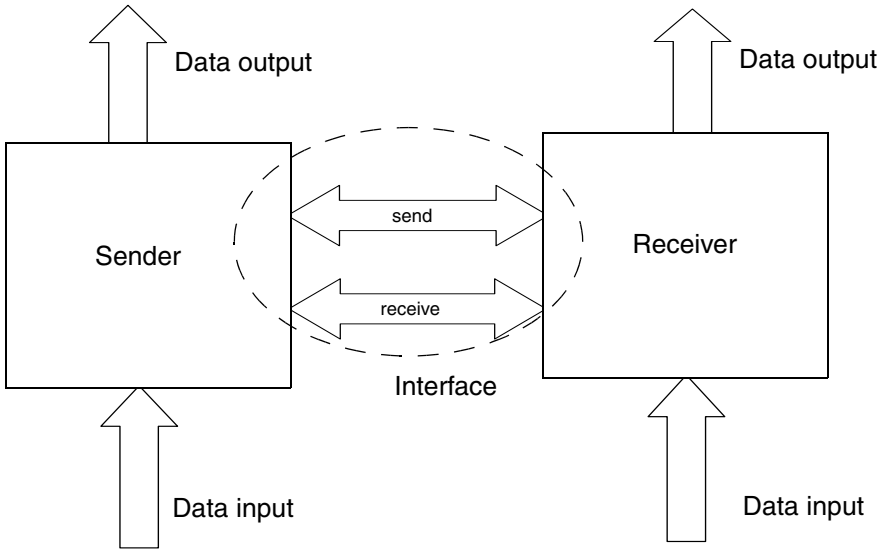
This chapter provides coding examples for using interface features in the following sections:

Basic Interface

A basic interface can be thought of as a bundle of bidirectional wires or nets connecting two or more modules. To understand how to create a basic interface, consider design feed_A, shown in Figure 7-1, which uses an interface for the send and receive buses. In this design:

- The Sender transmits its input to the Receiver through the interface.
- The Receiver then accepts the input from the Sender via the interface and outputs this data.
- The Receiver transmits its input to the Sender through the interface.
- The Sender then accepts the input from the Receiver via the interface and outputs this data.

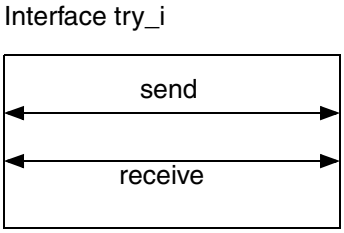
Figure 7-1 Design feed_A: Basic Interface



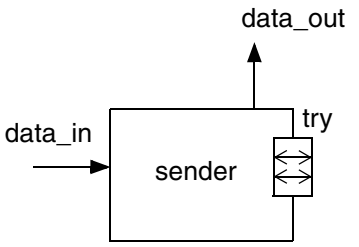
Example 7-1 shows the code for design feed_A, which has a basic interface that connects two modules.

Example 7-1 Design feed_A—Creating a Basic Interface

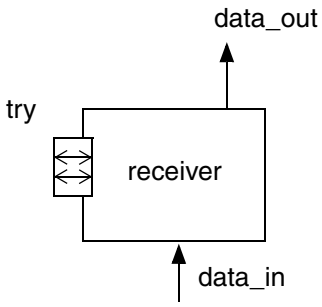
```
// interface definition

interface try_i;
  wire  [7:0] send, receive;
endinterface : try_i
```




```
// sender module definition

module sender(try_i  try,
              input logic [7:0] data_in,
              output logic [7:0] data_out );
  assign  data_out = try.receive;
  assign try.send = data_in;
endmodule
```



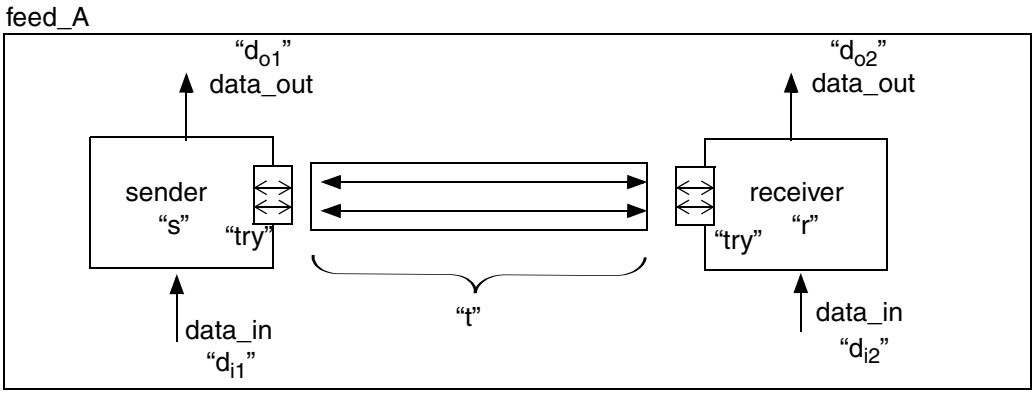

```
// receiver module definition

module receiver (try_i  try,
              input logic [7:0] data_in,
              output logic [7:0] data_out );
  assign data_out = try.send;
  assign try.receive = data_in;
endmodule
```




```
// top design definition

module feed_A (input wire [7:0] di1, di2,
               output wire [7:0] do1, do2);
  try_i    t();
  sender   s (t, di1, do1);
  receiver r (t, di2, do2);
endmodule
```

Figure 7-2 shows the interface block diagram.

Figure 7-2 Design feed_A: Interface Block Diagram



"t" is an instance of the interface try_i in module feed_A.
All signals in "t" and "try" are bidirectional.

The design feed_A is called a basic interface because it does not contain modports. For interfaces without modports, the default direction is inout for wires and ref for variables.

Note:

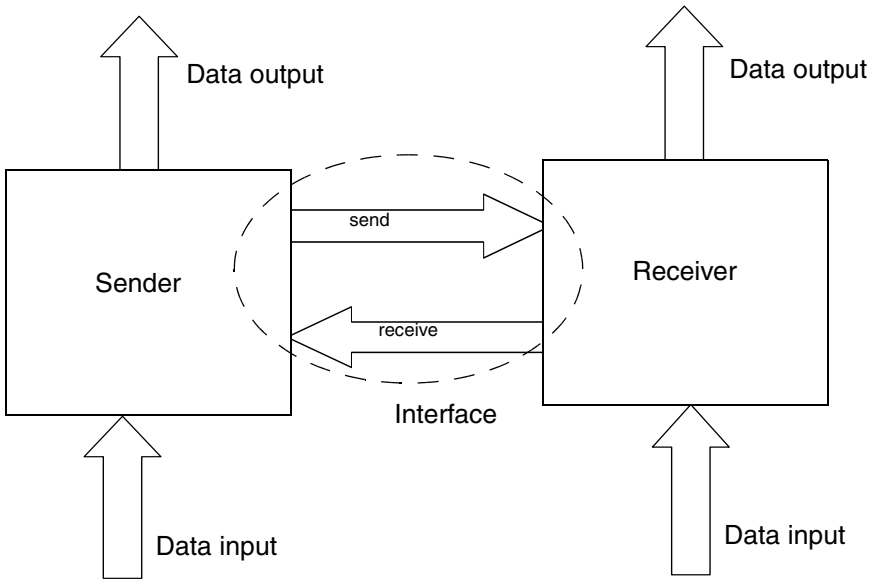
Variables referred to across an interface-type port are readable and writable. Therefore the semantics of access are like that of a module 'ref' port. In computer memory, this is similar to call by reference, where the last write wins. But in real hardware, this can only happen when modeled by the resolution semantics of wires. Because of this, ref ports are not realizable in silicon hardware, and hence not synthesizable. They are used only in simulation.

Only inouts declared as wires are synthesizable. In this example, the send and receive buses must be net type (wire) in order to be synthesized as inouts. If they are of any variable type (that is, reg, logic, or struct), they are not synthesizable unless modports are used. When creating interfaces, you should use modports to specify the direction of signals. This subject is described in the next section, "Interfaces With Modports" on page 7-4.

Interfaces With Modports

An interface with modports can be thought of as a bundle of wires or nets with a specified direction connecting the two or more modules. To understand how to create an interface with modports, consider design feed_B, shown in Figure 7-3. This design includes the functions of design feed_A and adds modports to the interface to specify the direction of the signals in the interface.

Figure 7-3 Design feed_B: Interface with Modports



Example 7-2 shows the interface definition of try_i. The interface try_i contains modports that can be used in sendmode or in receivemode. These modports enable you to specify signal direction when the interface is instantiated in modules.

Example 7-2 Interface Declaration of try_i: Direction of Send and Receive Buses Specified by modports sendmode and receivemode, Respectively

```
interface try_i;
  logic [7:0] send;
  logic [7:0] receive;

  modport sendmode   (output  send, input receive);
  modport receivemode(input  send, output receive);
endinterface
```

Figure 7-4 shows the block diagram for the interface try_i when used in sendmode by any module. Here, send is the output from the module and receive is the input to the module.

Figure 7-4 Interface try_i Block Diagram When Used in sendmode

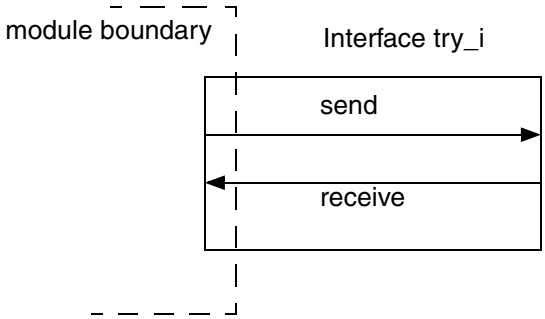
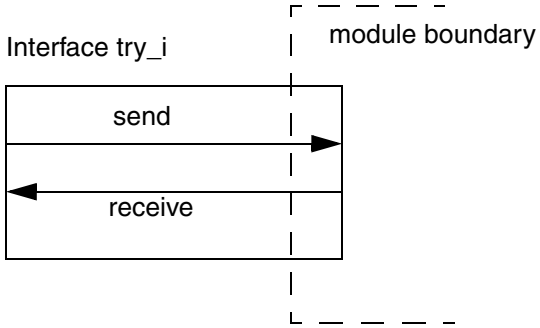


Figure 7-5 shows the block diagram for the interface try_i when used in receivemode by any module. Here, send is the input to the module and receive is the output from the module.

Figure 7-5 Interface try_i Block Diagram When Used in receivemode



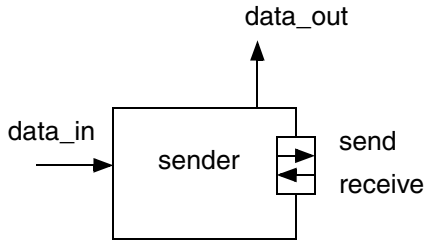
Example 7-3 shows the sender module definition. The sender module uses the sendmode modport of the interface try_i as a port named try. In the sendmode modport, the send bus is an output and the receive bus is an input.

Example 7-3 Sender Module With Modports in sendmode

```
module sender(try_i.sendmode  try,
              input logic [7:0] data_in,
                output logic [7:0] data_out );

  assign data_out = try.receive;

  assign try.send = data_in;

endmodule
```

Figure 7-6 shows the block diagram for the sender module, which uses the interface modport sendmode as one of its ports.

Figure 7-6 sender Module With Modport in sendmode



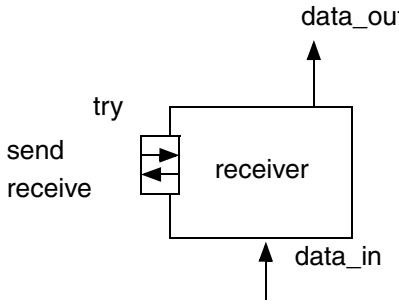
Example 7-4 shows the receiver module definition. The receiver module uses the receivemode modport of the interface try_i as a port named try. In the receivemode modport, the receive bus is an output and the send bus is an input.

Example 7-4 receiver Module With Modport in receivemode

```
module receiver(try_i.receivemode  try,
               input logic [7:0] data_in,
                output logic [7:0] data_out );

  assign data_out = try.send;

  assign try.receive = data_in;

endmodule
```

Figure 7-7 shows the block diagram for the receiver module, which uses the interface modport receivemode as one of its ports.

Figure 7-7 receiver Module With Modport in receivemode



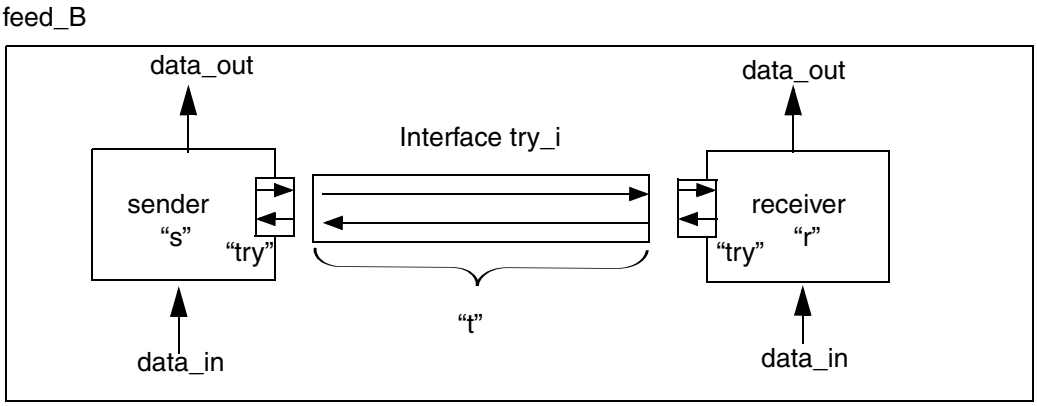
Example 7-5 shows the top-level design, feed_B, which instantiates the interface, t of type try_i, the sender module, s, and the receiver module, r.

Example 7-5 Design feed_B: Top Level

```
module feed_B (input wire [7:0] di1, di2, output wire [7:0] do1, do2);
  try_i    t();
  sender   s (t.sendmode, di1, do1);
  receiver r (t.receivemode, di2, do2);
endmodule
```

Figure 7-8 shows the block diagram for the top level of design feed_B.

Figure 7-8 Design feed_B: Top-Level Block Diagram



"t" is an instance of the interface try_i in module feed_B.
The receiver module uses the "receivemode" modport of the interface try_i as a port named "try".
The sender module uses the "sendmode" modport of the interface try_i as a port named "try".

Example 7-6 shows the entire feed_B design.

Example 7-6 Design feed_B: Interface, Sender, Receiver, and Top Level

```
interface try_i;
  logic [7:0] send;
  logic [7:0] receive;
  modport sendmode   (output  send, input receive);
  modport receivemode(input  send, output receive);
endinterface


module sender(try_i.sendmode  try,
              input logic [7:0] data_in,
            output logic [7:0] data_out );
   assign data_out = try.receive;
   assign try.send = data_in;
endmodule


module receiver(try_i.receivemode  try,
               input logic [7:0] data_in,
             output logic [7:0] data_out );
   assign data_out = try.send;
   assign try.receive = data_in;
endmodule


module feed_B(input wire [7:0] di1, di2, output wire [7:0] do1, do2);
   try_i    t();
   sender   s (t.sendmode, di1, do1);
   receiver r (t.receivemode, di2, do2);
endmodule
```

Interfaces With Functions

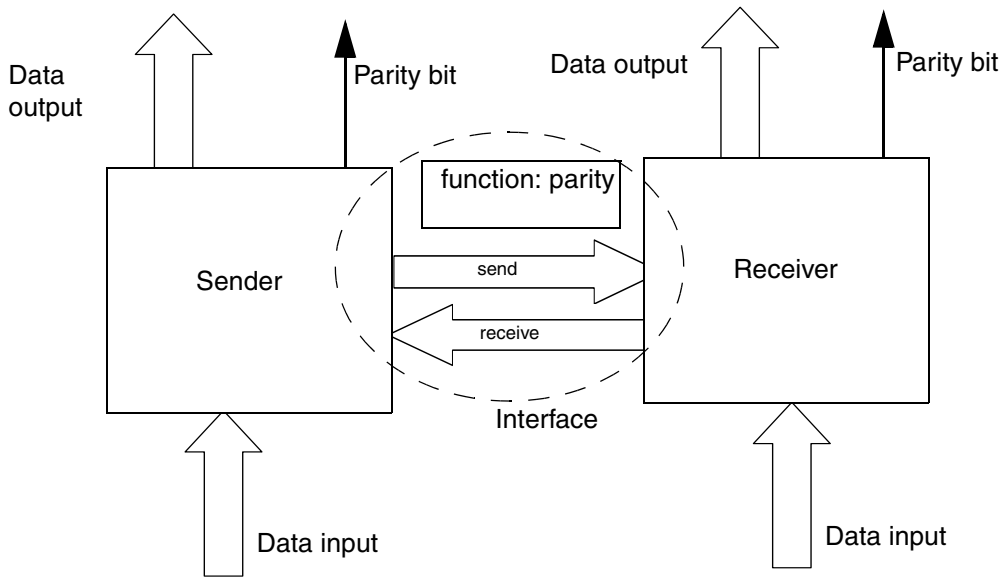
An interface can act as a single source for functions that are needed by modules. You can place functions inside interfaces and make these functions available to modules by using modports and the import keyword. It is important to understand that hardware is created only at the module that calls the function.

Note:

You can use functions in interfaces even if you don't have modports; however, the best practice is to use functions with modports.

To understand how to create an interface with modports and a function, consider design feed_C, shown in Figure 7-9. This design includes the functions of design feed_B and adds the parity function in the interface to determine parity.

Figure 7-9 Design feed_C: Interface With Modports and a Function



In design feed_C, both the sender and the receiver need the parity function in order to calculate the parity on the send and receive buses. You place the parity function in the interface and you import the parity function from the interface try_i via the sendmode and receivemode modports. This function then becomes available for the sender and receiver modules, which use the modports sendmode and receivemode, respectively. Note that modports are not required for importing the function but their use is recommended.

Example 7-7 shows the code for design feed_C and illustrates how to create an interface for the send and receive data buses with modports and a parity function.

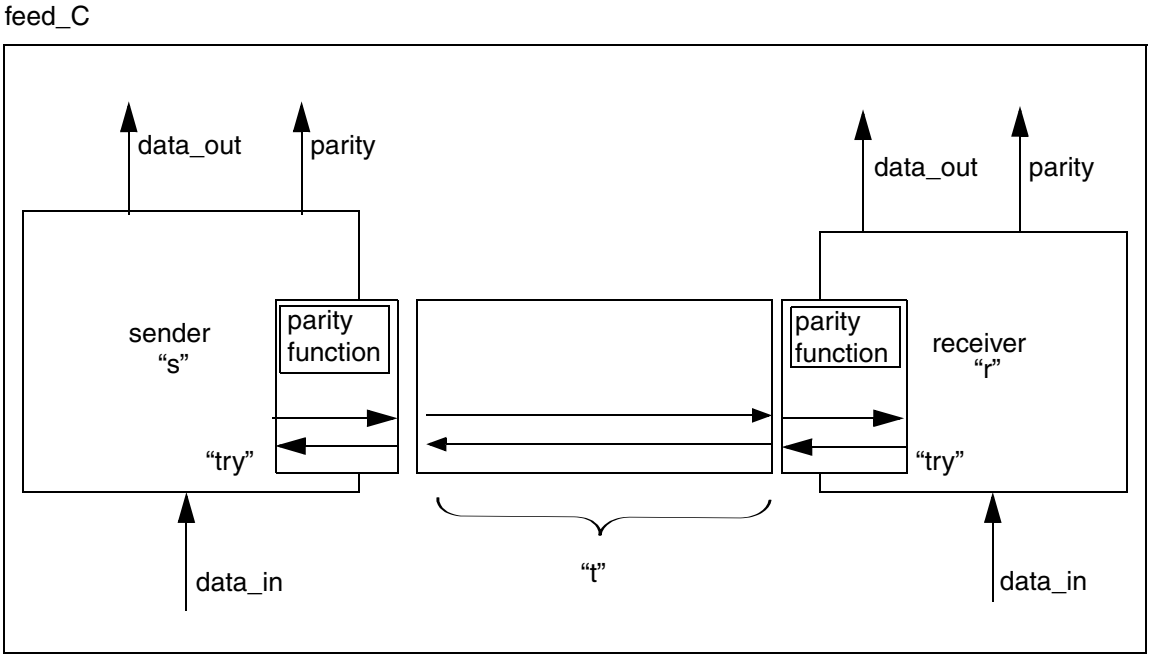
Example 7-7 Design feed_C: Inferring an Interface With Modports and a Function

```
interface try_i;
  logic [7:0] send;
  logic [7:0] receive;
  logic internal;

function automatic logic parity (logic [7:0] data);
  return(^data);
endfunction

  modport sendmode   (output send, input receive, import parity );
  modport receivemode (input send, output receive, import parity );

endinterface

module sender(try_i.sendmode  try,
              input logic [7:0] data_in,
               output logic [7:0] data_out,
```

```
                logic data_out_parity);

          assign data_out = try.receive;
          assign data_out_parity =  try.parity(try.receive);
        assign try.send = data_in;
endmodule

module receiver(try_i.receivemode  try,
              input logic [7:0] data_in,
                output logic [7:0] data_out,
                logic data_out_parity );
          assign data_out = try.send;
          assign data_out_parity =  try.parity(try.send);
        assign try.receive = data_in;
endmodule

module feed_C(input wire [7:0] di1, di2, output wire [7:0] do1, do2, logic p1, p2);
  try_i    t();
  sender   s (t.sendmode, di1, do1, p1);
  receiver r (t.receivemode, di2, do2, p2);
endmodule
```

Figure 7-10 shows the interface block diagram for design feed_C.

Figure 7-10 Design feed_C: Interface With Modports and a Function Block Diagram



“t” is an instance of the interface try_i in module feed_C.
The receiver module uses the “receivemode” modport of the interface try_i as a port named “try”.
The sender module uses the “sendmode” modport of the interface try_i as a port named “try”.
Both the sender and receiver modules import the parity function.

Interfaces With Functions and Tasks

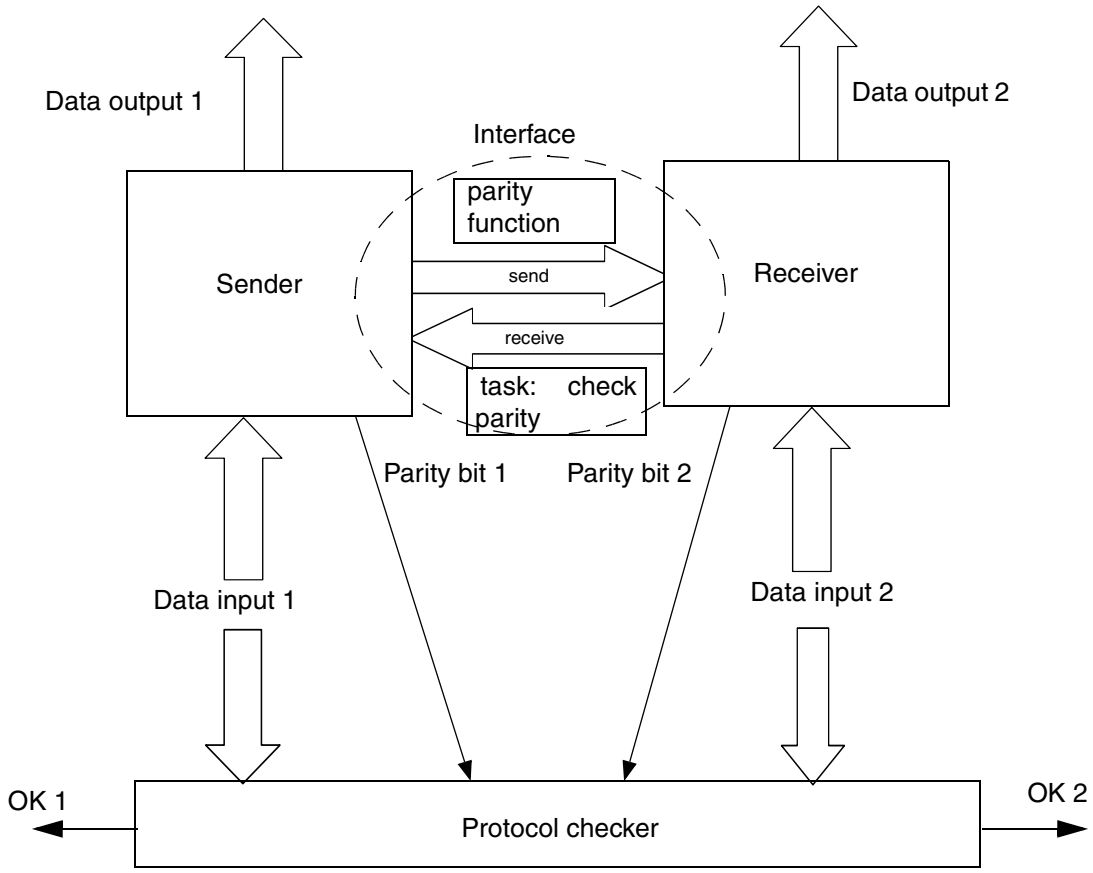
An interface can also act as a placeholder for tasks that are needed by modules. You can place tasks inside interfaces and make these tasks available to modules by using modports and the import keyword. It is important to note that hardware gets created only at the module that calls the tasks.

Note:

You can use functions and tasks in interfaces even if you don't have modports, but the best practice is to use them with modports.

To understand how to create an interface with a task, consider design feed_D, shown in Figure 7-11. This design includes the functions of design feed_C and adds a task that checks to see if the parity is correct.

Figure 7-11 Design feed_D: Interface With Modports, a Function, and a Task



Example 7-8 shows the code for design feed_D and illustrates how to create an interface for the send and receive data buses with modports, a function, and a task.

In Example 7-8, the module protocol_checker contains the task, parity_check, that checks to see if the parity is correct.

Example 7-8 Design feed_D: Creating an Interface With Modports, a Function, and a Task

```
interface try_i;
  logic [7:0] send;
  logic [7:0] receive;
  function automatic logic parity([7:0] data);
    return(^data);
  endfunction
  task automatic parity_check(input logic [7:0] data_sent, logic exp_parity,
                              output logic okay);
    if (exp_parity == ^data_sent) okay = '1;
    else okay = '0;
  endtask
  modport sendmode (output send, input receive, import parity );
  modport receivemode (input send, output receive, import parity );
  modport protocol_checkermode (import parity_check  );
endinterface

module sender(try_i.sendmode try, input logic [7:0] data_in, output logic [7:0]
data_out, logic data_out_parity);
  assign data_out = try.receive;
  assign data_out_parity = try.parity(try.receive);
  assign try.send = data_in;
endmodule

module receiver(try_i.receivemode try, input logic [7:0] data_in,
                output logic [7:0] data_out, logic data_out_parity);
  assign data_out = try.send;
  assign data_out_parity = try.parity(try.send);
  assign try.receive = data_in;
endmodule

module protocol_checker(input logic [7:0] data_sent, logic exp_parity,
                        output logic okay, try_i.protocol_checkermode try);
  always @ (data_sent) try.parity_check (data_sent, exp_parity, okay);

endmodule

module feed_D (input wire [7:0] di1, di2, output wire [7:0] do1, do2,
               logic p1, p2, okay1, okay2);
  try_i t();
  sender s (t.sendmode, di1, do1, p1);
  receiver r (t.receivemode, di2, do2, p2);
  protocol_checker pc1(di1, p2, okay1, t.protocol_checkermode);
  protocol_checker pc2(di2, p1, okay2, t.protocol_checkermode);
endmodule
```

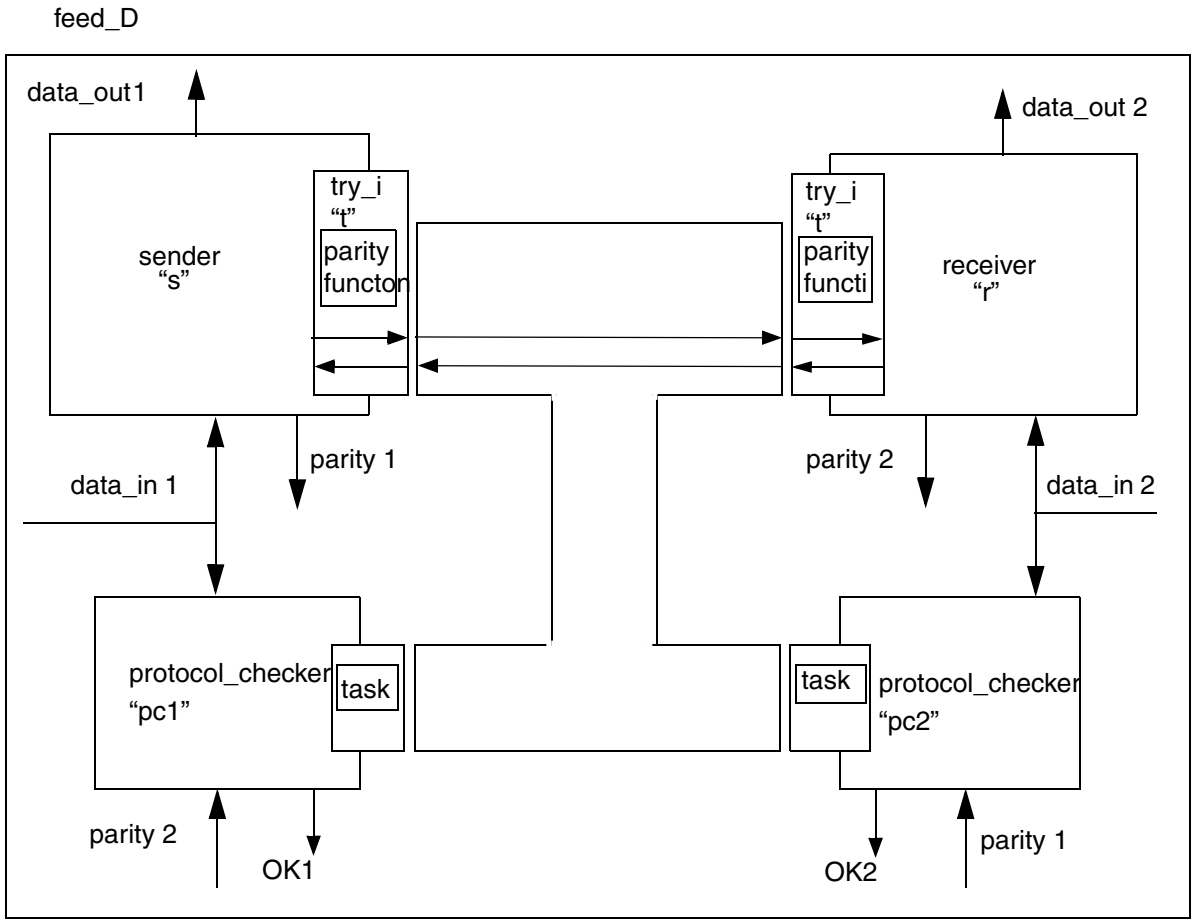
With the addition of the task, design feed_D contains the following three modports:

- The sendmode modport, which specifies the signal directions for the send and receive buses; this modport also imports the parity function. The sender module uses this modport.

- The receivemode modport, which specifies the signal directions for the send and receive buses; this modport also imports the parity function. The receiver module uses this modport.
- The protocol_checkermode modport, which imports the task parity_check. The two protocol_checker modules use this modport. These modules are instantiated as pc1 and pc2 inside design feed_D.

Figure 7-12 shows the interface block diagram.

Figure 7-12 Design feed_D: Interface With Modports, a Function, and a Task Block Diagram



Look at the design's gate-level netlist, shown in Example 7-11, and observe where the task hardware is created.

Example 7-9 GTECH Netlist

```
module sender_I_try_try_i_sendmode_ ( \try.send , \try.receive , data_in,
        data_out, data_out_parity );
  output [7:0] \try.send ;
```

```
  input [7:0] \try.receive ;
  input [7:0] data_in;
  output [7:0] data_out;
  output data_out_parity;
  wire   N0, N1, N2, N3, N4, N5;
  assign \try.send  [7] = data_in[7];
  assign \try.send  [6] = data_in[6];
  assign \try.send  [5] = data_in[5];
  assign \try.send  [4] = data_in[4];
  assign \try.send  [3] = data_in[3];
  assign \try.send  [2] = data_in[2];
  assign \try.send  [1] = data_in[1];
  assign \try.send  [0] = data_in[0];
  assign data_out[7] = \try.receive  [7];
  assign data_out[6] = \try.receive  [6];
  assign data_out[5] = \try.receive  [5];
  assign data_out[4] = \try.receive  [4];
  assign data_out[3] = \try.receive  [3];
  assign data_out[2] = \try.receive  [2];
  assign data_out[1] = \try.receive  [1];
  assign data_out[0] = \try.receive  [0];

  GTECH_XOR2 C7 ( .A(N5), .B(data_out[0]), .Z(data_out_parity) );
  GTECH_XOR2 C8 ( .A(N4), .B(data_out[1]), .Z(N5) );
  GTECH_XOR2 C9 ( .A(N3), .B(data_out[2]), .Z(N4) );
  GTECH_XOR2 C10 ( .A(N2), .B(data_out[3]), .Z(N3) );
  GTECH_XOR2 C11 ( .A(N1), .B(data_out[4]), .Z(N2) );
  GTECH_XOR2 C12 ( .A(N0), .B(data_out[5]), .Z(N1) );
  GTECH_XOR2 C13 ( .A(data_out[7]), .B(data_out[6]), .Z(N0) );
endmodule

module receiver_I_try_try_i_receivemode_ ( \try.send , \try.receive , data_in,
        data_out, data_out_parity );
  input [7:0] \try.send ;
  output [7:0] \try.receive ;
  input [7:0] data_in;
  output [7:0] data_out;
  output data_out_parity;
  wire   N0, N1, N2, N3, N4, N5;
  assign \try.receive  [7] = data_in[7];
  assign \try.receive  [6] = data_in[6];
  assign \try.receive  [5] = data_in[5];
  assign \try.receive  [4] = data_in[4];
  assign \try.receive  [3] = data_in[3];
  assign \try.receive  [2] = data_in[2];
  assign \try.receive  [1] = data_in[1];
  assign \try.receive  [0] = data_in[0];
  assign data_out[7] = \try.send  [7];
  assign data_out[6] = \try.send  [6];
  assign data_out[5] = \try.send  [5];
  assign data_out[4] = \try.send  [4];
  assign data_out[3] = \try.send  [3];
  assign data_out[2] = \try.send  [2];
  assign data_out[1] = \try.send  [1];
  assign data_out[0] = \try.send  [0];

  GTECH_XOR2 C7 ( .A(N5), .B(data_out[0]), .Z(data_out_parity) );
  GTECH_XOR2 C8 ( .A(N4), .B(data_out[1]), .Z(N5) );
```

```
  GTECH_XOR2 C9 ( .A(N3), .B(data_out[2]), .Z(N4) );
  GTECH_XOR2 C10 ( .A(N2), .B(data_out[3]), .Z(N3) );
  GTECH_XOR2 C11 ( .A(N1), .B(data_out[4]), .Z(N2) );
  GTECH_XOR2 C12 ( .A(N0), .B(data_out[5]), .Z(N1) );
  GTECH_XOR2 C13 ( .A(data_out[7]), .B(data_out[6]), .Z(N0) );
endmodule

module protocol_checker_I_try_try_i_protocol_checkermode_ ( data_sent,
        exp_parity, okay );
  input [7:0] data_sent;
  input exp_parity;
  output okay;
  wire   N0, N1, N2, N3, N4, N5, N6, N7, N8;

  GTECH_XOR2 C5 ( .A(exp_parity), .B(N1), .Z(N0) );
  GTECH_NOT I_0 ( .A(N0), .Z(N2) );
  GTECH_XOR2 C14 ( .A(N8), .B(data_sent[0]), .Z(N1) );
  GTECH_XOR2 C15 ( .A(N7), .B(data_sent[1]), .Z(N8) );
  GTECH_XOR2 C16 ( .A(N6), .B(data_sent[2]), .Z(N7) );
  GTECH_XOR2 C17 ( .A(N5), .B(data_sent[3]), .Z(N6) );
  GTECH_XOR2 C18 ( .A(N4), .B(data_sent[4]), .Z(N5) );
  GTECH_XOR2 C19 ( .A(N3), .B(data_sent[5]), .Z(N4) );
  GTECH_XOR2 C20 ( .A(data_sent[7]), .B(data_sent[6]), .Z(N3) );
  GTECH_BUF B_0 ( .A(N2), .Z(okay) );
endmodule

module feed_D ( di1, di2, do1, do2, p1, p2, okay1, okay2 );
  input [7:0] di1;
  input [7:0] di2;
  output [7:0] do1;
  output [7:0] do2;
  output p1, p2, okay1, okay2;

  wire   [7:0] \t.send ;
  wire   [7:0] \t.receive ;

  sender_I_try_try_i_sendmode_ s ( .\try.send (\t.send ), .\try.receive (
        \t.receive ), .data_in(di1), .data_out(do1), .data_out_parity(p1) );
  receiver_I_try_try_i_receivemode_ r ( .\try.send (\t.send ), .\try.receive (
        \t.receive ), .data_in(di2), .data_out(do2), .data_out_parity(p2) );
  protocol_checker_I_try_try_i_protocol_checkermode_ pc1 ( .data_sent(di1),
        .exp_parity(p2), .okay(okay1) );
  protocol_checker_I_try_try_i_protocol_checkermode_ pc2 ( .data_sent(di2),
        .exp_parity(p1), .okay(okay2) );
endmodule
```

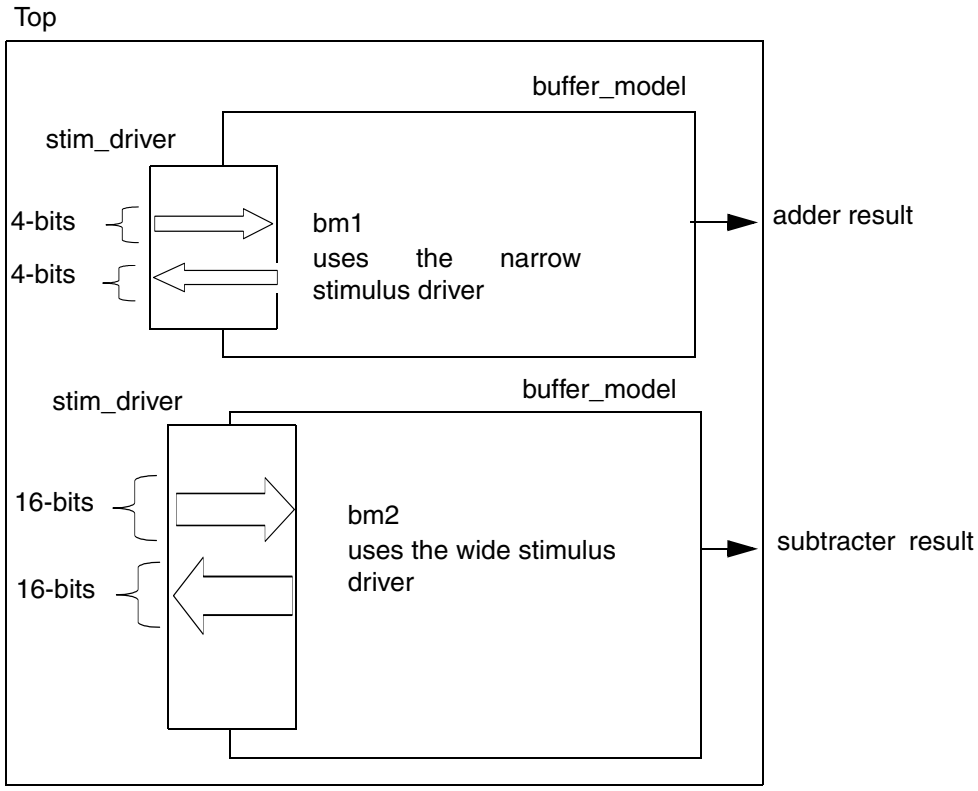
From the netlist shown in Example 7-9, you can see that the tool creates hardware at the site that called the task or function from the interface. In Example 7-7, the hardware for the parity functions was created in the sender and receiver modules (both were calling the function using the modports). Similarly in Example 7-8, the hardware for the task parity_check was created only in modules pc1 and pc2 (both were calling the task parity_check using the modports).

Parameterized Interfaces

Parameters can be used in interfaces just as you use parameters in modules. They can be modified on each instantiation of the interface.

To understand how to create parameterized interfaces, consider the design Top, shown in Figure 7-13. This design contains two stimulus drivers of different widths, one of 4-bit width and the other of 16-bit width.

Figure 7-13 Parameterized Interface



Example 7-10 shows you how to use parameters in an interface to create these different-size interfaces. The design builds two instances of the stim_driver interface. The narrow_stimulus_interface takes a parameter of four, and all the buses are 4-bits wide. The wide_stimulus_interface takes a parameter of 16, and all the buses are 16-bits wide inside the module top.

Example 7-10 Design Top, Using a Parameterized Interface

```
interface stim_driver;
  parameter BUS_WIDTH = 8;
  logic [BUS_WIDTH-1:0] sig1, sig2;
  function automatic logic parity_gen ([BUS_WIDTH-1:0] bus);
```

```
    return( ^bus );
  endfunction
  modport buffer_side (input sig1,output sig2,import parity_gen);
endinterface

module buffer_model #(parameter DELAY =1)(stim_driver.buffer_side a,
                      output logic par_rslt);
   always
   begin
     a.sig2 = #DELAY ~a.sig1;
     par_rslt =  a.parity_gen(a.sig2);
   end
endmodule

module top # (parameter WIDTH1 = 4, WIDTH2 = 16)(output logic pr1, pr2);
  stim_driver #(WIDTH1)narrow_stimulus_interface();
  stim_driver #(WIDTH2)wide_stimulus_interface();
  buffer_model bm1(narrow_stimulus_interface.buffer_side, pr1);
  buffer_model bm2(wide_stimulus_interface.buffer_side, pr2);
endmodule
```

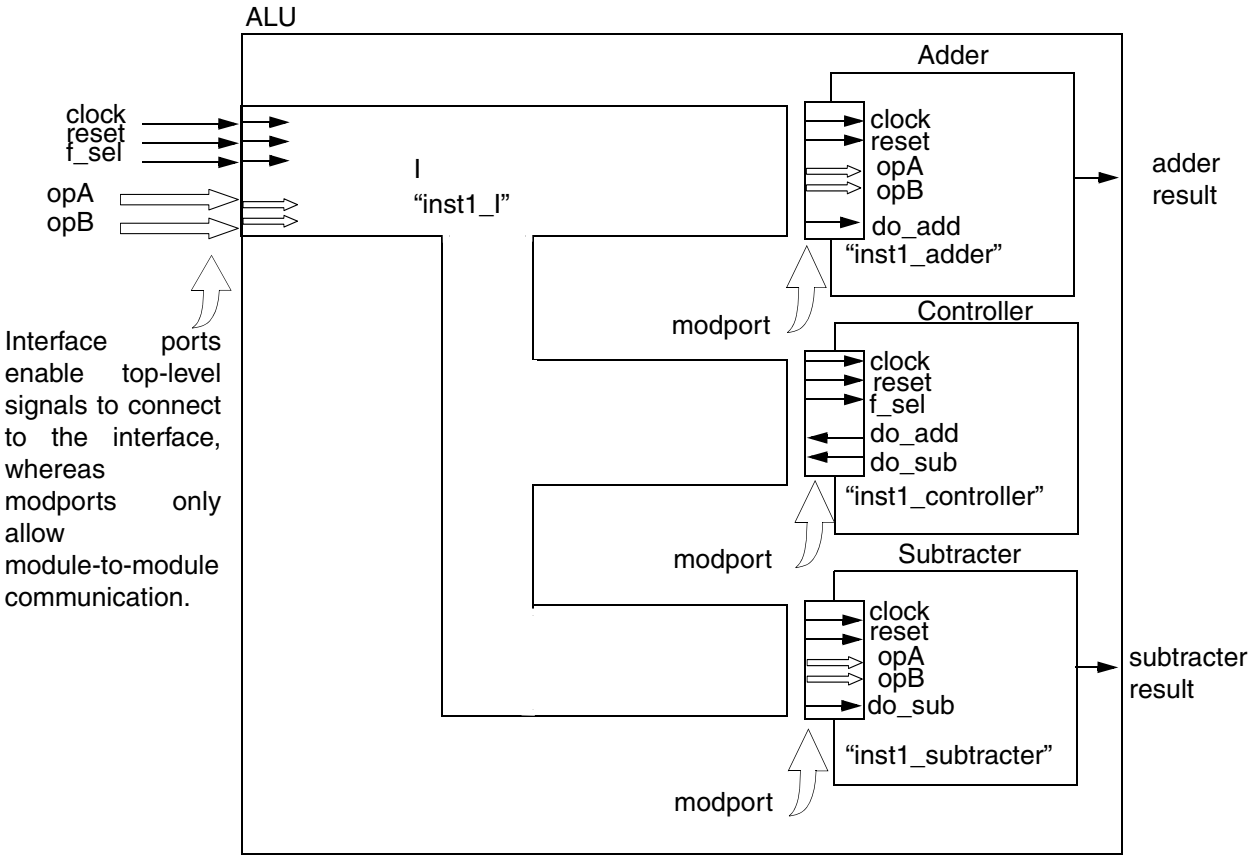
For more information about linking large interface-based designs, see “Hierarchical Elaboration Flow for SystemVerilog Designs With Interfaces” on page 1-9.

Ports in Interfaces

In all of the interface examples discussed in sections “Basic Interface” on page 7-2 through “Parameterized Interfaces” on page 7-17, an interface connects intermodule communication signals. Ports in an interface allow you to connect an external net or variable to the interface, which in turn connects them to the lower-level modules as needed. Once the external signals are inside the interface, you can connect these signals to modules that instantiate the interface. To understand the concept of a port in an interface, consider the design ALU, shown in Figure 7-14. The design ALU uses an interface to connect all the top-level inputs to the subdesigns. In design ALU

- clock, reset, f_sel, opA and opB are the top-level input ports
- adder_result and subtracter_result are the output ports

Figure 7-14 Design ALU: Creating Ports in an Interface



Note: The signals do_add and do_sub are declared inside the interface.
The adder module uses the adder_mp modport of the interface I.
The subtracter module uses the subtracter_mp modport of the interface I.
The controller module uses the controller_mp modport of the interface I.

Top-level signals are shared through the ports in the interface. The ALU design creates interface ports for the signals clock, reset, f_sel, opA, and opB by including them in the port list of the interface as shown below:

```
interface I (input logic clock, reset, f_sel, logic [7:0] opA, opB);
```

The adder module uses the global signals clock, reset, opA, and opB and the local signal do_add. These signals are specified in the adder_mp modport as follows:

```
modport adder_mp (input clock, reset, do_add, opA, opB);
```

The adder_mp modport is used by the adder module.

The subtracter module uses the global signals clock, reset, opA, and opB and the local signal do_sub. These signals are specified in the subtracter_mp modport as follows:

```
modport subtracter_mp (input clock, reset, do_sub, opA, opB);
```

The subtracter_mp modport is used by the subtracter module.

The controller module uses the global signals clock, reset, and f_sel and the local signals do_add and do_sub. These signals are specified in the controller_mp modport as follows:

```
modport controller_mp (input clock, reset, f_sel, output do_add, do_sub);
```

The controller_mp modport is used by the controller module.

In summary, there are three modports inside the interface declaration:

- adder_mp modport
- subtracter_mp modport
- controller_mp modport

The do_add and do_sub signals are intermodule communication signals and are not interface ports. These signals are local to the interface and declared as

- logic do_add;
- logic do_sub;

inside the interface I declaration.

The entire code for the ALU design is shown in Example 7-11.

Example 7-11 ALU Design: Creating Ports in an Interface

```
interface I (input logic clock, reset, f_sel, logic [7:0] opA, opB);
   logic do_add;
   logic do_sub;
   modport adder_mp (input clock, reset, do_add, opA, opB);
   modport subtracter_mp (input clock, reset, do_sub, opA, opB);
   modport controller_mp (input clock, reset, f_sel, output do_add, do_sub);
endinterface : I

module adder(I.adder_mp adder_signals, output logic [7:0] sum);
  always_ff @(posedge adder_signals.clock, negedge adder_signals.reset)
    if (!adder_signals.reset) sum = '0;
    else if (adder_signals.do_add) sum = adder_signals.opA + adder_signals.opB;
endmodule : adder

module subtracter(I.subtracter_mp sub_signals, output logic [7:0] difference);
  always_ff @(posedge sub_signals.clock, negedge sub_signals.reset)
    if (!sub_signals.reset) difference = '0;
    else if (sub_signals.do_sub) difference = sub_signals.opA + sub_signals.opB;
endmodule : subtracter

module controller(I.controller_mp controller_signals);
```

```
  always_ff @(posedge controller_signals.clock, negedge controller_signals.reset)
    begin
      if (!controller_signals.reset)
        begin
          controller_signals.do_add = '0;
          controller_signals.do_sub = '0;
          end
            else if (~controller_signals.f_sel) //decode logic
               begin
               controller_signals.do_add = '1;
               controller_signals.do_sub = '0;
               end
            else if (controller_signals.f_sel)
               begin
               controller_signals.do_add = '0;
               controller_signals.do_sub = '1;
               end
    end
endmodule : controller

module alu(input clock, reset, f_sel, [7:0] opA, opB, output [7:0] adder_result,
           subtracter_result);
    I inst1_I(.clock, .reset, .f_sel, .opA, .opB);
    adder inst1_adder(inst1_I.adder_mp, adder_result);
    subtracter inst1_subtracter(inst1_I.subtracter_mp, subtracter_result);
    controller inst1_controller(inst1_I.controller_mp);
endmodule : alu
```

always Blocks in Interfaces

The always block is supported within an interface, as illustrated in Example 7-12.

Example 7-12 Flip-Flop in an Interface

```
interface I(input clk, rst, d, output logic q);

always_ff @(posedge clk, negedge rst)
    if (!rst)
      q <= 0;
    else
      q <= d;

endinterface: I

module top(input clock, reset, data_in, output q_out);

I inst1(clock, reset, data_in, q_out);

endmodule : top
```

In Example 7-12 there is a description of a flip-flop inside the interface I. When the interface is instantiated in the module top, a D flip-flop is created as shown in the inference report as follows:

```
============================================================================
|    Register Name    |   Type    | Width | Bus | MB | AR | AS | SR | SS | ST |
============================================================================
|     inst1.q_reg     | Flip-flop |   1   |  N  | N  | Y  | N  | N  | N  | N  |
============================================================================
```

Renaming Conventions

Modules can have parameters, interfaces, and interface modports as ports. Also, interfaces can use parameters. These various connecting methods affect the module port names in the GTECH and gate-level netlist.

The format the tool uses for renaming the module ports is as follows:

```
modulename_p1v1_p2v2...I_portname_interfacename_modportname_vi1_vi2...
```

The format is described in Table 7-1.

Table 7-1 Renaming Format For GTECH Netlist

Item	Description
modulename	Name of the module
p1	First parameter name inside the module
v1	Value of the first parameter
p2	Second parameter name inside the module
v2	Value of the second parameter
This convention continues if there are additional parameters.	
I	Indicates an interface
portname	Name of the port inside the module that uses the interface as a port
interfacename	Name of the interface used in the module port
modportname	Name of the modport used with the interface as a module port
vi1	Value of the first parameter

Table 7-1 Renaming Format For GTECH Netlist

Item	Description
vi2	Value of the second parameter
And so on, if there are more parameters.	

Renaming Example 1

To understand the renaming convention described in Table 7-1, consider design feed_A in Example 7-13. (Note: design feed_A is discussed earlier in the chapter in "Basic Interface" on page 7-2 and repeated here.)

Example 7-13 Design feed_A

```
interface try_i;
  wire  [7:0] send, receive;
endinterface : try_i

module sender(try_i  try,
              input logic [7:0] data_in,
              output logic [7:0] data_out );
  assign  data_out = try.receive;
  assign try.send = data_in;
endmodule

module receiver(try_i  try,
              input logic [7:0] data_in,
              output logic [7:0] data_out );
  assign data_out = try.send;
  assign try.receive = data_in;
endmodule


module feed_A (input wire [7:0] di1, di2, output wire [7:0] do1, do2);
  try_i    t();
  sender   s (t, di1, do1);
  receiver r (t, di2, do2);
endmodule
```

The renaming data for the sender module is the following:

```
modulename : sender
p1: none
v1:none
I : indicates interface
portname: try
interfacename: try_i
modportname : none
pi1: none
vi1: none
```

Based on the above data, the tool renames module sender to module sender_I_try_try_i_.

A portion of the GTECH netlist is shown in Example 7-14 to illustrate the renamed module.

Example 7-14 GTECH Netlist

```
module sender_I_try_try_i__ ( try_send, try_receive, data_in, data_out );
  inout [7:0] try_send;
  inout [7:0] try_receive;
  input [7:0] data_in;
.....
```

The renaming data for the receiver module is the following:

```
modulename : receiver
p1: none
v1:none
I : indicates interface
portname: try
interfacename: try_i
modportname : none
pi1: none
vi1: none
```

Based on the above data, the tool renames module receiver to module receiver_I_try_try_i__.

A portion of the GTECH netlist is shown in Example 7-15 to illustrate the renamed module.

Example 7-15 Module receiver GTECH Netlist

```
module receiver_I_try_try_i__ ( try_send, try_receive, data_in, data_out );
  inout [7:0] try_send;
  inout [7:0] try_receive;
  input [7:0] data_in;
.....
```

Renaming Example 2

To understand the renaming convention described in Table 7-1 on page 7-22, consider design feed_B in Example 7-16. (Note: design feed_B is discussed in Example 7-6 on page 7-9 and repeated here.)

Example 7-16 Design feed_B

```
interface try_i;
  logic [7:0] send;
  logic [7:0] receive;
  modport sendmode   (output  send, input receive);
  modport receivemode (input  send, output receive);
endinterface


module sender(try_i.sendmode  try, input logic [7:0] data_in,
               output logic [7:0] data_out );
```

```
   assign data_out = try.receive;
   assign try.send = data_in;
endmodule

module receiver(try_i.receivemode  try, input logic [7:0] data_in,
                 output logic [7:0] data_out );
   assign data_out = try.send;
   assign try.receive = data_in;
endmodule


module feed_B (input [7:0] di1, di2, output [7:0] do1, do2);
   try_i    t();
   sender   s (t.sendmode, di1, do1);
   receiver r (t.receivemode, di2, do2);
endmodule
```

The renaming data for the sender and receiver modules follows:

```
modulename : sender
p1: none
v1:none
I : indicates interface
portname: try
interfacename: try_i
modportname : sendmode
pi1: none
vi1: none

modulename : receiver
p1: none
v1:none
I : indicates interface
portname: try
interfacename: try_i
modportname : receivemode
pi1: none
vi1: none
```

Based on the above data, the tool renames modules sender and receiver to sender_I_try_try_i_sendmode_ and receiver_I_try_try_i_receivemode_, respectively.

A portion of the GTECH netlist is shown in Example 7-17 to illustrate the renamed modules.

Example 7-17 GTECH Netlist

```
module sender_I_try_try_i_sendmode_ ( try_send, try_receive, data_in,
data_out );
  output [7:0] try_send;
  input  [7:0] try_receive;
  input  [7:0] data_in;

.....
module receiver_I_try_try_i_receivemode_ ( try_send, try_receive,
data_in, data_out );
  input  [7:0] try_send;
  output [7:0] try_receive;
  input  [7:0] data_in;
.....
```

Renaming Example 3

To understand the renaming convention described in Table 7-1 on page 7-22, consider the design in Example 7-18. (Note: this design is discussed earlier in the chapter in Example 7-10 and repeated here.)

Example 7-18 RTL Design

```
interface stim_driver;
  parameter BUS_WIDTH = 8;
  logic [BUS_WIDTH-1:0] sig1, sig2;
  function automatic logic parity_gen ([BUS_WIDTH-1:0] bus);
    return( ^bus );
  endfunction
  modport buffer_side (input sig1,output sig2,import parity_gen);
endinterface

module buffer_model #(parameter DELAY =1)(stim_driver.buffer_side a,
                      output logic par_rslt);
   always
   begin
     a.sig2 = #DELAY ~a.sig1;
     par_rslt =  a.parity_gen(a.sig2);
   end
endmodule

module top # (parameter WIDTH1 = 4, WIDTH2 = 16)(output logic pr1, pr2);
  stim_driver #(WIDTH1)narrow_stimulus_interface();
  stim_driver #(WIDTH2)wide_stimulus_interface();
  buffer_model bm1(narrow_stimulus_interface.buffer_side, pr1);
  buffer_model bm2(wide_stimulus_interface.buffer_side, pr2);
endmodule
```

The renaming data for the modules follows:

```
modulename : buffer_model
```

```
p1: DELAY
v1:1   (default value)
I : indicates interface
portname: a
interfacename: stim_driver
modportname : buffer_side
pi1: BUS_WIDTH
vi1: 4 (explicit modification)


modulename : buffer_model
p1: DELAY
v1:1   (default value)
I : indicates interface
portname: a
interfacename: stim_driver
modportname : buffer_side
pi1: BUS_WIDTH
vi1: 16 (explicit modification)
```

Based on the above data, the tool renames bm1 to

```
buffer_model_DELAY1_I_a_stim_driver_buffer_side_BUS_WIDTH4_
```

and the tool renames bm2 to

```
buffer_model_DELAY1_I_a_stim_driver_buffer_side_BUS_WIDTH16_
```

Interface Restrictions

This section describes synthesis interface restrictions.

1. Interfaces must be defined first and used later. You should include the interface definition and the design to be compiled that is using the interface in the same compilation unit. If the interface and the design are in the same file, the interface must be defined first. However, if the interface and the design are defined in different files, do the following:

   - Include the interface definition at the beginning of the design file by using the `include construct.
   - Analyze the interface and the design files with the `analyze` command, placing the interface file before the design file.

   For example, the code in Example 7-19 is allowed because interface I is declared first and then later used in the module buffer as a port. Conversely, the code in Example 7-20 is not supported because interface I is first used in the buffer module and defined later.

Example 7-19

```
interface I;
```

```
  logic in, out;
  modport MP(input in, output out);
endinterface : I

module buffer(I.MP a);
  assign a.out = a.in;
endmodule : buffer
```

Example 7-20

```
module buffer(I.MP a);
  assign a.out = a.in;
endmodule : buffer

interface I;
  logic in, out;
  modport MP(input in, output out);
endinterface : I
```

2. In an interface or module instantiation, interface instances cannot be connected to modport ports of lower-level modules. Example 7-21 illustrates unsupported usage. Within module top, the module instance B1 connects the interface instance, inst1, to the modport port x of module bottom. This is not allowed in synthesis.

Example 7-21

```
interface I;
  logic a1;
  logic b1;

  logic a2;
  logic b2;

  modport a1_to_b1(input a1, output b1);
  modport a2_to_b2(input a2, output b2);

  modport b1_to_a1(output a1, input b1);
  modport b2_to_a2(output a2, input b2);

endinterface

module bottom(I.a1_to_b1 x);
  assign x.b1 = x.a1;
endmodule

module top;
  I inst1();
  bottom B1(.x(inst1));
endmodule
```

The code in Example 7-21 is not supported, but the opposite construction, shown in Example 7-22, is allowed. Here, within module top, the module B1 instance connects the interface modport inst1.a1_to_b1 to the interface port x in the module bottom.

Example 7-22

```
interface I;
  logic a1;
  logic b1;

  logic a2;
  logic b2;

  modport a1_to_b1(input a1, output b1);
  modport a2_to_b2(input a2, output b2);

  modport b1_to_a1(output a1, input b1);
  modport b2_to_a2(output a2, input b2);

endinterface


module bottom(I x);
  assign x.b1 = x.a1;

endmodule


module top;
  I inst1();
  bottom B1(.x(inst1.a1_to_b1));
endmodule
```

3. Interfaces must only contain purely automatic tasks and functions unless the interface variables used by the interface method are listed in the modport or the interface instance is in the same module that is calling the interface method. Otherwise, the tool returns an error message similar to the following:

   ```
   Error:  ...: Static function 'adder' is not synthesizable
   in $unit, expecting "automatic" keyword (VER-523)
   ```

   By default, the tool assumes a static function and returns a VER-523 error message unless you use the keyword automatic.

4. Exporting of tasks and functions from one module into an interface is not supported for synthesis.

5. External fork and join constructs in interfaces are not supported for synthesis.

6. When variables (such as logic and struct) are used inside interfaces with no modports, they are treated as ref type and are not synthesizable. For example, if the elements send and receive from the first interface in Example 7-1 are defined as logic, the design will not be synthesizable.

8

Other Coding Styles

This chapter provides examples of supported SystemVerilog features in the following sections:

- Casting
- Predefined SYSTEMVERILOG Macro
- Using Matching Block Names
- Multidimensional Arrays
- Port Renaming in Multidimensional Arrays

Assignment Patterns

The support for assignment patterns is controlled by the `hdlin_sv_ieee_assignment_patterns` variable. The default value of this variable is 1. At this setting, the tool supports all IEEE-1800 SystemVerilog assignment patterns in the right-hand side of all legal, synthesizable assignment-like contexts, except module and interface instantiations.

If you want to include assignment patterns for module instantiations (assignment patterns on ports), set `hdlin_sv_ieee_assignment_patterns` to the value 2. Note that your runtime may take twice as long when you set `hdlin_sv_ieee_assignment_patterns` to 2. Due to the cross-module nature of pattern assignments on ports, you can only use this feature in the analyze/elaborate flow; the `read_sverilog` command is not supported.

Under default conditions (that is, when `hdlin_sv_ieee_assignment_patterns` is 1), the tool supports assignment patterns when context type is known, as shown in the following example:

```
typedef struct {int i; int j; logic [0:1] l;} T;
module single (input int a, b, output T y);
  assign y = '{int: a+b, default: 0};
endmodule
```

In the context of this module, the type of 'T' is known.

When you set `hdlin_sv_ieee_assignment_patterns` to 2, the tool supports assignment patterns when context type is not known. For example,

```
typedef struct {int i; int j; logic [0:1] l;} T;

module top (input int a, b);
// context type 'T' not known
down inst1 ('{int: a+b, default: 0});
endmodule

module down (input T in);
// …
endmodule
```

In the context of the top module, the type 'T' is not explicitly known, but the tool can infer the correct assignment. This extra processing increases runtime.

Assignment Pattern Restrictions:

- The tool does not support assignment patterns on the left-hand side of assignments.
- The tool does not support assignment patterns on output ports of module instantiations.

- The tool does not support the 'var' keyword.
- The tool does not support array pattern keys in assignment patterns.

To help understand the difference between assignment patterns and concatenation, consider the following example:

```
typedef struct packed {int field;} T;
module test(output test_bit);
  T v0, v2;
  assign v0 =  {1'b1 + 1'b1}; // v0 = 0
  assign v2 = '{1'b1 + 1'b1}; // v2 = 2
  assign test_bit = ((v0==0)&&(v2==2));
endmodule
```

In this example, the v0 assignment uses concatenation while the v2 assignment uses assignment patterns `{}. For more information on assignment patterns, refer to the IEEE Standard for SystemVerilog (IEEE 1800-2005).

Macro Expansion and Parameter Substitution

In SystemVerilog, macro expansion happens well before parameters get substituted. Therefore, when you use parameters in macro calls, you can expect the parameter names to remain even after the macro call. To help understand this process, consider Example 8-1.

Example 8-1 Understanding Macro Expansion

```
 `define MAC(x) P``x
module test #(parameter N = 2, PN = 4, P2 = 8)(output test_bit);
 assign test_bit = (`MAC(N) == PN);
endmodule
```

This code results in test_bit having a value of 1. Because macro expansion occurs before parameter substitution, when the macro MAC is called by `MAC(N), it gets replaced by PN. Therefore, since the line (PN == PN) is true, test_bit gets a value of 1. If parameter substitution occurred before macro expansion, N would get a value of 2, resulting in MAC(2) being P2, resulting in 8. In this case, test_bit would be 0, because 8 is not equal to 4.

‘begin_keywords and ‘end_keywords

To prevent errors on legacy code that contains SystemVerilog keywords, wrap the code in the `‘begin_keywords` and `‘end_keywords` constructs as shown in Example 8-2. Identify the appropriate code standard by using `version_specifier`.

Note:
Values for `version_specifier` are: 1364-1995,1364-2001, 1364-2001-noconfig, 1364-2005, and 1800-2005.

The code in Example 8-2 was not originally written for SystemVerilog, so it uses “logic” as the name of an output. However, when this code is repurposed for SystemVerilog, you must wrap the code with the `‘begin_keywords` and `‘end_keywords` because “logic” is a keyword in SystemVerilog; otherwise the tool reports an error.

Example 8-2 ‘begin_keywords and ‘end_keywords Example

```
`begin_keywords "1364-2005"
module test (input a, input b , output logic);
  assign logic = a | b;
endmodule
`end_keywords
```

You can only specify these constructs outside of a design element (module, primitive, configuration, interface, program, or package). The `` `begin_keywords `` directive affects all source code that follows the directive, even across source code file boundaries, until the matching `` `end_keywords `` directive is encountered.

Tasks and Functions - Binding by .name

The tool supports binding tasks and functions by the .name style as shown in Example 8-3.

Example 8-3

```
module test(input integer value1, value2, output logic [32:0] result1, result2);
  function logic [32:0] adder (integer a, b);
    return (a + b);
  endfunction
  //Pass by order style:
  assign result1 = adder(value1,value2);
  //.name style:
  assign result2 = adder(.b(value2), .a(value1));
endmodule
```

Functions with default argument values are not supported. The following code is not supported:

```
function logic [32:0] adder33 (int a, b = 25);
return (a + b);
endfunction
```

For the code above, the tool issues the following VER-721 error :

```
Error:  ..../task_binding/test1_error/test1.sv:3: The construct 'default
expression for an argument' is not supported.  (VER-721)
```

Note:
Per the LRM: “If both positional and named arguments are specified in a single subroutine call, then all the positional arguments must come before the named arguments.”

Automatic Variable Initialization

By default, the tool supports initialization of automatic variables. If there is no explicit initialization, the tool initializes automatic variables with a default value of ‘0’. This is the same for both 2-state and 4-state variables.

Variables in for Loops

In Verilog, each for loop needs its own loop index, as shown in Example 8-4. SystemVerilog allows you to use a single variable, as shown in Example 8-5.

Example 8-4

```
module varloop1 (input clk, input [3:0] in, output reg [3:0] out);
  integer j;
  integer k;
  reg [3:0] tmp;
  always @(posedge clk) begin
    for(j=0;j<4;j=j+1) begin
      tmp[j] = !in[j];
    end
  end
  always @(posedge clk) begin
    for(k=0;k<4;k=k+1) begin
      out[k] <= tmp[k];
    end
  end
endmodule
```

In Example 8-5, the variable, j, is used as the loop index both loops.

Example 8-5

```
module varloop (input clk, input [7:0] in, output logic [3:0] out);
  logic [7:0] tmp;
  always_ff @(posedge clk) begin
    for(int j=0;j<8;j=j+2) begin
      tmp[j]  = !in[j];
      tmp[j+1]= in[j+1];
    end
  end
  always_ff @(posedge clk) begin
    for(int j=0;j<4;j++) begin
```

```
      out[j] <= tmp[j];
    end
  end
endmodule
```

Bit-Level Support for Compiler Directives

The following compiler directives can be applied at the bit level:

- `async_set_reset`, see Example 8-6
- `sync_set_reset`, see Example 8-7 and Example 8-8
- `one_hot`, see Example 8-9
- `one_cold`, see Example 8-10
- `keep_signal_name`

The following examples show usage.

Example 8-6 async_set_reset

```
module dff_async (input clk, d, st, rst, output logic q);
// synopsys async_set_reset "st, rst"
always_ff @ (posedge clk or posedge st or posedge rst)
begin
  if (st)       q <= 1'b1;
  else if (rst) q <= 1'b0;
  else          q <= d;
end
endmodule
```

Example 8-7 sync_set_reset

```
module dff_sync (input clk, d, st, rst, output logic q);
// synopsys sync_set_reset rst
always_ff @ (posedge clk)
begin
  if (rst) q <= 1'b0;
  else     q <= d;
end
endmodule
```

Example 8-8 sync_set_reset

```
module dff_sync (input clk, d, st, rst, output logic q);
// synopsys sync_set_reset "st rst"
always_ff @...
```

Example 8-9 one_hot

```
module dff_async (input clk, d, st, rst, output logic q);
// synopsys one_hot "st, rst"
```

```
always_ff @...
```

Example 8-10 one_cold

```
module dff_async (input clk, d, st, rst, output logic q);
// synopsys one_cold "st, rst"
always_ff @...
```

Structures

The Synopsys synthesis tool supports the SystemVerilog data type called struct. By using the struct data type, you can group a collection of variables into a single object.

Example 8-11 shows how to declare a structured data type. Here, a CPU instruction, which consists of an 8-bit opcode and a 32-bit address, is represented as a data type called instruction. IR1, IR2, IR3 are three new variables that are of the type instruction. Example 8-11 also uses the typedef construct, which allows multiple references to the same data type.

Example 8-11 Declaring a Structured Data Type

```
typedef struct {
byte opcode;     // 8 bits
int addr;        // 32 bits
} instruction;   // named structure type

module m;
instruction IR1, IR2, IR3; // define variable
//...
endmodule
```

Example 8-12 uses packed structures, enums, and $unit. Even though the code is written using high-level abstraction with features such as typedef, enums, $unit, and structures, the synthesis tool builds a 35-bit register that consists of asynchronous reset flip-flops.

Example 8-12 Using Packed Structures

```
typedef enum logic [1:0] {OFF = 2'd0 , ON = 2'd3} SWITCH_VALUES;

//default - integer for enums
// RED = 0, GREEN = 1, BLUE = 2
typedef enum {RED, GREEN, BLUE} LIGHT_COLORS;

typedef struct packed {
SWITCH_VALUES switch;  //  2 bits
LIGHT_COLORS light;    // 32 bits
logic test_bit;        //  1 bit
} sw_lgt_pair;

module struct_default(input logic clock, reset, output sw_lgt_pair slp);
  always_ff @ (posedge clock, posedge reset)
```

```
  begin
    if(reset)
      //clears all 35 bits since packed structure ( 35 bit vector)
      slp <= '0;
    else
    begin
      slp.switch <= ON;
      slp.light <= GREEN;
      slp.test_bit <= 1;
    end
  end
endmodule
```

In Example 8-12, all the members of the structure are reset to 0 by the following line:

```
if(reset)  slp <= '0;
```

It is not necessary to initialize each member of the structure because it is packed. The tool treats packed structures as a single vector as shown in the inference report in Example 8-13.

Example 8-13 Inference Report

```
=============================================================================
|    Register Name    |   Type    | Width | Bus | MB | AR | AS | SR | SS | ST
=============================================================================
|       slp_reg       | Flip-flop |  35   |  Y  | N  | Y  | N  | N  | N  | N
=============================================================================
Presto compilation completed successfully.
```

However, for unpacked structures, each member needs to be initialized separately, or the tool reports an error. Consider Example 8-14, which uses an unpacked structure.

Example 8-14 Incorrect Initialization for an Unpacked Structure

```
struct_literal_new_unpacked_error.sv
typedef enum  logic [1:0] {ON = 2'd3  , OFF = 2'd0} SWITCH_VALUES;
typedef enum {RED, GREEN, BLUE} LIGHT_COLORS;//default - integer for
enums

typedef struct {
 SWITCH_VALUES switch;//2 bits
 LIGHT_COLORS light;//32 bits
 logic test_bit;// 1 bit
    } sw_lgt_pair;


module struct_default(input logic clock, reset, output sw_lgt_pair slp);

   always_ff @(posedge clock, posedge reset)
      begin
         if(reset)  slp <= '0;
         else
            begin
               slp.switch <= OFF;
               slp.light <= GREEN;
               slp.test_bit <= 1;
            end
      end
endmodule
```

In Example 8-14, the code incorrectly initializes the unpacked structure in the line

```
if(reset)  slp <= '0;
```

The tool does not allow initialization of all the members of the structure and reports an error, as shown below:

```
Compiling source file ..../structures/
struct_literal_new_unpacked_error.sv
Error:  .../structures/struct_literal_new_unpacked_erro
r.sv:15: non-equivalent types in assignment operation.
(ELAB-930)
*** Presto compilation terminated with 1 errors. ***
```

For unpacked structures, you must initialize each member of the structure separately. This is done using this line of code:

```
if(reset)  slp <= '{SWITCH_VALUES : ON,  LIGHT_COLORS : RED, logic :
```

The entire example is shown in Example 8-15.

Example 8-15 Initialization Is Successful

```
typedef enum  logic [1:0] {ON = 2'd3  , OFF = 2'd0} SWITCH_VALUES;
typedef enum {RED, GREEN, BLUE} LIGHT_COLORS;//default - integer for
enums
```

```
typedef struct {
 SWITCH_VALUES switch;//2 bits
 LIGHT_COLORS light;//32 bits
 logic test_bit;// 1 bit
    } sw_lgt_pair;


module struct_default(input logic clock, reset, output sw_lgt_pair slp);

   always_ff @ (posedge clock, posedge reset)
      begin
      if(reset)  slp <= '{SWITCH_VALUES : ON,  LIGHT_COLORS : RED, logic
: 1'b0};
      // since it is an unpacked structure you need to initialize each
member
         else
         begin
            slp.switch <= OFF;
            slp.light <= GREEN;
            slp.test_bit <= 1;
         end
      end
endmodule
```

Example 8-15 builds a 2-bit register using asynchronous set flip-flops, and a 33-bit register using asynchronous reset flip-flops, as shown in the inference report in Example 8-16.

Example 8-16 Inference Report

```
============================================================================
|    Register Name    |   Type    | Width | Bus | MB | AR | AS | SR | SS | ST
|==========================================================================
|       slp_reg       | Flip-flop |   2   |  Y  | N  | N  | Y  | N  | N  | N
|       slp_reg       | Flip-flop |  33   |  Y  | N  | Y  | N  | N  | N  | N
============================================================================
Presto compilation completed successfully.
```

Port Renaming in Structures

When structures are used in module ports, the names of the ports are modified in the resulting netlist. To understand the renaming convention, consider Example 8-17. When building the netlist, the tool renames the ports as shown in Example 8-18.

Example 8-17

```
typedef struct {
logic [3:0] field;    //       4 bits
logic flag;           //       1 bit
logic [6:0] result;   //       7 bits
} packet;             //total 12 bits

module test(input packet p1, output packet p2);
  assign p2 = p1;
endmodule
```

The GTECH netlist, containing the renamed ports, is shown in Example 8-18.

Example 8-18

```
module test ( .p1({\p1[field][3] , \p1[field][2] , \p1[field][1] ,
        \p1[field][0] , \p1[flag] , \p1[result][6] , \p1[result][5] ,
        \p1[result][4] , \p1[result][3] , \p1[result][2] , \p1[result][1] ,
        \p1[result][0] }), .p2({\p2[field][3] , \p2[field][2] , \p2[field][1] ,
        \p2[field][0] , \p2[flag] , \p2[result][6] , \p2[result][5] ,
        \p2[result][4] , \p2[result][3] , \p2[result][2] , \p2[result][1] ,
        \p2[result][0] }) );
  input \p1[field][3] , \p1[field][2] , \p1[field][1] , \p1[field][0] ,
        \p1[flag] , \p1[result][6] , \p1[result][5] , \p1[result][4] ,
        \p1[result][3] , \p1[result][2] , \p1[result][1] , \p1[result][0] ;
  output \p2[field][3] , \p2[field][2] , \p2[field][1] , \p2[field][0] ,
        \p2[flag] , \p2[result][6] , \p2[result][5] , \p2[result][4] ,
        \p2[result][3] , \p2[result][2] , \p2[result][1] , \p2[result][0] ;
  wire   \p2[field][3] , \p2[field][2] , \p2[field][1] , \p2[field][0] ,
        \p2[flag] , \p2[result][6] , \p2[result][5] , \p2[result][4] ,
        \p2[result][3] , \p2[result][2] , \p2[result][1] , \p2[result][0] ;
  assign \p2[field][3]  = \p1[field][3] ;
  assign \p2[field][2]  = \p1[field][2] ;
  assign \p2[field][1]  = \p1[field][1] ;
  assign \p2[field][0]  = \p1[field][0] ;
  assign \p2[flag]  = \p1[flag] ;
  assign \p2[result][6]  = \p1[result][6] ;
  assign \p2[result][5]  = \p1[result][5] ;
  assign \p2[result][4]  = \p1[result][4] ;
  assign \p2[result][3]  = \p1[result][3] ;
  assign \p2[result][2]  = \p1[result][2] ;
  assign \p2[result][1]  = \p1[result][1] ;
  assign \p2[result][0]  = \p1[result][0] ;

endmodule
```

Unions

The synthesis tool supports the SystemVerilog union construct. Example 8-19 shows construct usage and Example 8-20 shows the inference report.

For synthesis union restrictions, see Appendix B, “Unsupported Constructs.”

Example 8-19 union Construct

```
typedef struct {
  union packed{
    logic [31:0] data;
    int i;
  }ff;
} my_struct;

module union_example(input clk, input my_struct d, output my_struct q);
  my_struct loop_index;
  always_ff@(posedge clk)
  begin
    for (loop_index.ff.i = 0; loop_index.ff.i <= 31; loop_index.ff.i++)
      q.ff.data[loop_index.ff.i] <= d.ff.data[loop_index.ff.i];
  end
endmodule
```

Example 8-20 Inference Report

```
============================================================================
|    Register Name    |   Type    | Width | Bus | MB | AR | AS | SR | SS | ST |
============================================================================
|       q_reg         | Flip-flop |   32  |  Y  | N  | N  | N  | N  | N  | N  |
============================================================================
```

Port Renaming in Unions

In SystemVerilog, if packed unions with same-sized members are used in module ports, the names of the ports are modified in the resulting netlist. There is no visibility to the different members of the union. In Example 8-21, field1 and field2 are not visible in the gate-level or GTECH netlist. They are just 7-bit vectors of p1 and p2.

To understand the renaming convention, consider Example 8-21. When building the netlist, the tool renames the ports as shown in Example 8-22.

Example 8-21 RTL

```
typedef union packed {
logic [7:0] field1;
byte field2;
} packet;

module test(input packet p1, output packet p2);
   assign p2.field2 = p1.field1;
endmodule
```

The resulting netlist with renamed ports is shown in Example 8-22.

Example 8-22 Netlist

```
module test ( p1, p2 );
  input [7:0] p1;
  output [7:0] p2;

  assign p2[7] = p1[7];
  assign p2[6] = p1[6];
  assign p2[5] = p1[5];
  assign p2[4] = p1[4];
  assign p2[3] = p1[3];
  assign p2[2] = p1[2];
  assign p2[1] = p1[1];
  assign p2[0] = p1[0];

endmodule
```

Implicit Instantiation of Ports

The Synopsys synthesis tool supports implicit instantiation of ports using the .name syntax or the .* syntax.

This capability eliminates the Verilog 2001 requirement of listing a port name twice when the port name and signal name are identical, while still listing all of the ports of the instantiated module. This new way of connecting port names applies to both module ports and ports in an interface.

Example 8-23 shows .name usage.

Example 8-23 .name Usage

```
module dot_name(input in, clk, rst, output logic out);
  dff U1(.in, .clk, .rst, .out);
endmodule
module dff (input in, clk, rst, output logic out);
  always_ff @(posedge clk or negedge rst)
  if(!rst) out <= '0;
  else out     <= in;
endmodule
```

In Example 8-23, the module dff has port names in, out, clk, and rst. This module is instantiated inside module dot_name, whose port names are also the same.

SystemVerilog provides a concise way to connect the ports of the dot_name module to the ports of dff, using the statement

```
dff U1(.in, .clk, .rst, .out);
```

This is equivalent to the Verilog style of doing port connections as shown below:

```
dff U1(.in(in), .clk(clk), .rst(rst), .out(out));
```

Note:

You can mix the Verilog 2001 style and SystemVerilog style when you connect the module ports. That is, if the dff module has a port named reset instead of rst, the following is allowed:

```
dff U1(.in, .clk, .reset(rst), .out);
```

Example 8-24 shows the full example.

Example 8-24 Mixing Verilog 2001 and SystemVerilog Styles

```
module dot_name (input in, clk, rst, output logic out);
  dff U1(.in, .clk, .reset(rst), .out );
endmodule
module dff(input in, clk, reset, output logic out);
  always_ff @(posedge clk or negedge reset)
  if(!reset) out <= '0;
  else       out <= in;
endmodule
```

Notice that ports in, out, and clk have the same instance port name and module port name, but the reset signal is different. Using the .* port connection style, you can write the RTL code more compactly as

```
dff U1(.*, .reset(rst));
```

When you use the .* style, the tool connects all the ports where the instance port name and size and the module port name and size are the same.

Example 8-25 shows the full example.

Example 8-25 . Port Connection Usage*

```
module dot_star_test(input in, clk, rst, output logic out);
  dff U1(.*, .reset(rst));
endmodule
module dff(input in, clk, reset, output logic out);
  always_ff @(posedge clk or negedge reset)
  if(!reset) out <= '0;
  else out       <= in;
endmodule
```

The .* port connection style can also be applied to interface ports. When you use the .* style, you must analyze all your lower-level modules before you elaborate, or the tool reports an ELAB-397 error. See “Port Connection (.*) Reading Restriction” on page 9-21.

Ports in interfaces can also be connected using the .name construct as shown in Example 8-26.

Example 8-26 dot_name_interface.sv

```
interface I(input logic clk, rst, logic [7:0] d, output logic [7:0] q);
endinterface

module M(I i1);
endmodule

module dot_name(input logic clk, rst, logic [7:0] d, output logic [7:0]
q);

  I i1(.rst, .clk, .q, .d); // OLD WAY I i1(.rst(rst), .clk(clk), .q(q),
.d(d))
  M m1(.i1);  //OLD WAY  M m1(.inst1(i1)); if module declaration M had I
inst1

endmodule
```

In Example 8-26, the interface instance name is i1 in the module M declaration. The interface instantiation name is also i1 inside module dot_name. The synthesis tool supports the SystemVerilog construct .name, which allows the code to be written as below

```
M m1(.i1);
```

which is equivalent to the Verilog 2001 style

```
M m1(.i1(i1));
```

Using the .name syntax enables you to simplify coding port connections and to compact your code.

Functions and Tasks

This section contains examples that use the following SystemVerilog task and function features:

- Void functions
- All types as legal task and function argument types
- All types as legal function return types

- Return statement in functions
- Logic default task and function argument type
- Input default task and function argument direction

In Example 8-27, default input and logic are used in the adder33 function to build a 33-bit adder.

Example 8-27 33-Bit Adder

```
module add_func_new(input logic [31:0] val1, val2, output logic [32:0] result);
  function  logic [32:0] adder33 ([31:0] val1, val2);// default input and logic
    return (val1 + val2);
  endfunction
   assign result = adder33(val1, val2);
endmodule
```

Example 8-28 uses explicit longint arguments, val1 and val2, and returns a value to and from subtractor64.

Example 8-28 64-Bit Subtracter

```
module subtractor_func_longint_new (input longint val1, val2, output longint result);
  function  longint subtractor64 (longint val1, val2); // default input direction
   return( val1 - val2 );
  endfunction
  assign result = subtractor64 (val1, val2);
endmodule
```

In Example 8-29, the typedef construct enables you to create user-defined structures in module ports, task inputs, and output arguments. This code builds an adder and a subtracter.

Note:
Struct ports are expanded out as vectors in gate-level netlists as follows:

```
RTL                 BECOMES    GATES
input_vals.val_1               val_1_and_val_2[val_1][31:0]
input_vals.val_2                val_1_and_val_2[val_2][31:0]
addsub.sum                     result[sum][32:0]
addsub.diff                     result[diff][31:0]
```

Example 8-29 Adder and Subtracter

```
typedef struct {
reg [32:0] sum;
reg [31:0] diff;
} addsub;
typedef struct {
reg [31:0] val_1;
reg [31:0] val_2;
} input_vals;
module struct_to_and_from_task (input input_vals  val_1_and_val_2,
                                output addsub result);
    task calc_values (input  input_vals val_1_and_val_2,  output  addsub result);
```

```
     addsub tmp;
     tmp.sum = val_1_and_val_2.val_1 +  val_1_and_val_2.val_2;
     tmp.diff =  val_1_and_val_2.val_1 -  val_1_and_val_2.val_2;
     result =  tmp;
   endtask
   always_comb calc_values (val_1_and_val_2, result);
endmodule
```

In Example 8-30, a function returns a value and an output argument. This design has a subtracter and a multiplier in one function.

Example 8-30 Subtracter and Multiplier in One Function

```
module function_with_output_arguments #(parameter N=8)(input logic
 [N-1:0] A, B, output logic [N:0] DIFF, logic [2*N-1:0] PROD);
  function automatic logic [N-1:0]  prod_and_diff (input logic [N-1:0]
                                                   Aval, Bval, output
  logic [2*N-1:0] prod_val);
    prod_val = Aval * Bval;
    return (Aval - Bval);
  endfunction

  always_comb
    DIFF = prod_and_diff(A, B, PROD);

endmodule
```

Example 8-31 counts the number of zeros in the input data_stream and outputs the result. In this example, the following two automatic functions are declared in $unit:

- Function legal: Checks if the pattern is legal using the SystemVerilog style of for loops.
- Function zeros: Counts the number of zeros. This function has the output argument, num_zeros, and returns nothing; therefore, the function is a pseudo void function. It also uses the SystemVerilog style of for loops.

Both functions use autoassignment ++ operators.

Example 8-31

```
function automatic logic legal (input [7:0] x);
  reg seenZero, seenTrailing;
  begin :_legal_block
    legal = 1; seenZero = 0; seenTrailing = 0;
    for(int i = 0; i <= 7; i++)
      if( seenTrailing && (x[i] == 1'b0) )
      begin
        return 0;
      end
      else if( seenZero && (x[i] == 1'b1) )
        seenTrailing = 1;
      else if( x[i] == 1'b0 )
      seenZero = 1;
  end
endfunction

function automatic void zeros ([7:0] data_stream, output logic [3:0] num_zeros);
```

```
  logic [3:0] count;
  count = 0;
  for(int i = 0; i <= 7; i++)
    if( data_stream[i] == 1'b0)
      count++;
  num_zeros = count;
endfunction

module count_zeros (input logic [7:0] data_stream,
                    output logic [3:0] result, logic error);
  wire is_legal = legal(data_stream);
  logic [3:0] temp_result;
  assign error =! is_legal;
  always_comb zeros(data_stream, temp_result);
  assign result = is_legal ? temp_result : 1'b0;
endmodule
```

Specifying Data Types for Parameters

You can specify the data type for a parameter by using the following keywords:

- parameter
- type

This allows the data type to be modified for parameters on an instance-by-instance basis for modules as well as interfaces. If you do not specify a data type for a parameter, the type defaults to type logic.

Example 8-32 through Example 8-34 illustrate how you specify the type for parameters by using the parameter type keyword. Examples include

- Both user-defined data types and standard data types such as int, shortint, longint, and reg
- Both module instances and interface instances

To understand construct usage in standard data types and module instances, consider Example 8-32.

Example 8-32 Construct Usage in Standard Data Types and Module Instances

```
module comparator #(parameter type comparatortype = int)
   (input comparatortype a, comparatortype b,output logic lt, logic gt, logic eq);
  always_comb
  begin
    unique if (a < b)
    begin
      lt = 1'b1;
      gt = 1'b0;
      eq = 1'b0;
    end
    else if (a > b)
```

```
    begin
      lt = 1'b0;
      gt = 1'b1;
      eq = 1'b0;
    end
    else if ( a == b)
    begin
      eq = 1'b1;
      lt = 1'b0;
      gt = 1'b0;
    end
  end
endmodule

module top(input int a1, b1, shortint a2, b2, longint a3, b3, output logic [2:0]
 less_than, greater_than, equal);

//32-bit comparator
comparator comparator32 (a1, b1, less_than[0], greater_than[0], equal[0]);

//16-bit comparator
comparator #(.comparatortype(shortint))
             comparator16 (a2, b2, less_than[1], greater_than[1], equal[1]);

//long comparator
comparator #(.comparatortype(longint))
             comparator_fp (a3, b3, less_than[2], greater_than[2], equal[2]);
endmodule
```

In Example 8-32, the module comparator has two inputs, a and b, which are of int data type, and three outputs, lt, eq, gt, which are of logic data type. This is specified in the source code as

```
module comparator #(parameter type comparatortype = int)
(input comparatortype a, comparatortype b,
output logic lt, logic gt, logic eq);
```

The module top has three instances of the comparator but with different data types.

- comparator32 defines inputs a1 and b1 with the default int data type; the comparator module accepts the int data type.

  ```
  //32-bit comparator
  comparator comparator32(a1, b1, less_than[0],greater_than[0],
  equal[0]);
  ```

- comparator16 defines inputs a2 and b2 with the nondefault data type, shortint; therefore, explicit data type redefinition is needed. This is accomplished with the following code:

  ```
  //16-bit comparator
  comparator #(.comparatortype(shortint))
  comparator16(a2,b2, less_than[1], greater_than[1], equal[1]);
  ```

- comparator_fp defines inputs a3 and b3 with the nondefault data type, longint; therefore, explicit data type redefinition is needed. This is done with the following code:

```
//long comparator
comparator #(.comparatortype(longint))
comparator_fp(a3,b3, less_than[2], greater_than[2], equal[2]);
```

To understand how to use parameters with user-defined data types, consider Example 8-33.

Example 8-33 Using Parameters With User-Defined Data Types

```
typedef struct {
  logic [31:0] src_a;
  logic [31:0] dst_a;
  logic  [3:0] hdr;
} data_packet;


typedef struct {
  logic [63:0] src_a;
  logic [63:0] dst_a;
  logic  [9:0] hdr;
} big_data_packet;


module test #(parameter type data_packet_type = data_packet)
      (input data_packet_type a, output data_packet_type b);
   assign b = a;
endmodule

module top(input data_packet di1, output data_packet do1,
          input big_data_packet bdi1, output big_data_packet bdo1);
   test u1(di1, do1);
   test #(.data_packet_type(big_data_packet)) u2(bdi1, bdo1);
endmodule : top
```

In Example 8-33, two data types are user-defined:

- data_packet

```
typedef struct {
  logic [31:0] src_a;
  logic [31:0] dst_a;
  logic  [3:0] hdr;
} data_packet;
```

- big_data_packet

```
typedef struct {
  logic [63:0] src_a;
  logic [63:0] dst_a;
  logic  [9:0] hdr;
} big_data_packet;
```

Module test uses the data type data_packet to define the ports input a and output b. This is specified in the source code as

```
module test #(parameter type data_packet_type = data_packet)
              (input data_packet_type a, output data_packet_type b);
    assign b = a;
endmodule
```

Module top defines two instances of module test and uses data_packet for one instance and big_data_type for the other instance.

- u1 uses the default data type of data_packet for a and b. Therefore no redefinition is required and the declaration is

  ```
  test u1(di1, do1);
  ```

- u2 uses the nondefault data type, big_data_type, for a and b; this requires an explicit redefinition, which is done by the following code:

  ```
  test #(.data_packet_type(big_data_packet)) u2(bdi1, bdo1);
  ```

  Note:
  data_packet and big_data_packet are structures defined by using typedef in $unit so that they can be used in module ports of u1 and u2.

To understand how to use parameters in interfaces, consider Example 8-34.

Example 8-34 Using Parameters in Interfaces

```
interface I # (parameter type new_type = logic)(input new_type clk, rst,
d);
   modport MP(input clk, rst, d);
endinterface : I

module dff(I.MP a, output logic q);
   always_ff @ (posedge a.clk, negedge a.rst)
     begin
        if(!a.rst) q <= '0;
        else q <= a.d;
     end
endmodule : dff


module two_dff (input clk, rst, d [1:0], output logic q [1:0]);

   I inst1 (.clk, .rst, .d(d[1]));//two state
   dff u1(.a(inst1.MP), .q(q[1]));

   I # (.new_type(logic)) inst2 (.clk, .rst, .d(d[0]));//four state
   dff u2 (.a(inst2.MP), .q(q[0]));
endmodule : two_dff
```

In Example 8-34, the module two_ff has two interfaces. In the interface inst1(), all the ports are of bit type, the default type. Therefore, no explicit redefinition is needed; the source code contains

```
I inst1(.clk, .rst, .d(d[1]), .q(q[1]));
```

In the second interface, inst2(), all the ports are of logic type, a nondefault type. This requires an explicit redefinition. This is done with the following source code line:

```
I #(.new_type(logic))inst2(.clk, .rst, .d(d[0]), .q(q[0]))
```

Casting

The tool supports the SystemVerilog casting construct. Example 8-35 shows its usage and illustrates the following casting features:

- Size casting
- Sign casting
- Casting to a user-defined type called my_struct

Example 8-35 also uses the $bits system task to determine the size of a packed array by computing the total number of bits in the packed array of my_struct inside struct_array_in.

Example 8-35 SystemVerilog casting Construct Usage

```
localparam VEC = 1;
localparam STRUCT_ARRAY_SIZE = 2;

typedef logic [3:0] nibble;

typedef enum nibble {A=1, B=2, C=4, D=8} one_hot_variable;


typedef struct packed{
one_hot_variable  [VEC:0] nibble_array; // 2 * 4 = 8 bits
logic b;         // 1 bit
} my_struct;// total 9 bits

module test(
    input my_struct [STRUCT_ARRAY_SIZE:0] struct_array_in,   // 9 * 3 = 27 bits
    output logic [$bits(struct_array_in)-1:0] packed_array,  // 27 bits
    output my_struct single_struct,                          // 9 bits
    output logic [19:0] twenty_bits_of_packed_array,// 20 bit
    output logic one_bit_of_packed_array_with_sign//signed 1 bit
    );

// assigns the entire array of packed structures
// to a packed vector of equivalent size
    assign packed_array = struct_array_in;

// assigns the members of struct_array_in[0] single_struct by using
```

```
// casting on user defined type my_struct
   assign single_struct = my_struct'(packed_array);

// size casting, assigning 20 bits of the packed array
   assign twenty_bits_of_packed_array = 20'(packed_array);

// sign casting
   assign one_bit_of_packed_array_with_sign =
   signed'(twenty_bits_of_packed_array);

endmodule
```

Predefined SYSTEMVERILOG Macro

The synthesis tool supports the predefined SYSTEMVERILOG macro. Example 8-36 shows usage.

Example 8-36 Predefined SYSTEMVERILOG Macro Usage

```
`ifdef SYSTEMVERILOG
module M(input logic i, output int o);
`else
module M(input i, output signed [31:0] o);
`endif
//...
endmodule
```

In this way, you can insert SystemVerilog constructs into your existing Verilog code. All the predefined macros for Verilog 2005 are defined in SystemVerilog.

Using Matching Block Names

SystemVerilog adds the capability of allowing a matching block name after the end keyword. This enhances the readability of the code, quickly letting users know which block of code is being grouped. The matching block name is optional and can be applied to endinterface, endmodule, endtask, endfunction, and named begin-end blocks.

To understand matching block name usage in a state machine, consider Example 8-37, which uses named blocks to code an FSM. Here, the combinational sections of the code in the always_comb block have the block name combinational at the begin-end code blocks. The sequential sections of the code in the always_ff block have the block name sequential at the begin-end code blocks. Lastly, at the end of the module drink_machine, there is the drink_machine block name, indicating the end of the module.

Example 8-37 Using Matching Block Names in a State Machine

```
`define vend_a_drink do {D, dispense, collect} = {IDLE,2'b11}; while(0)

module drink_machine (input logic nickel_in, dime_in, quarter_in, reset,
```

```
clk,
                     output logic collect, nickel_out, dime_out,
dispense);

   localparam IDLE=0, FIVE=1, TEN=2, TWENTY_FIVE=3, FIFTEEN=4, THIRTY=5,
              TWENTY=6, OWE_DIME=7;

   logic [2:0] D, Q;

   always_comb
      begin : combinational
         nickel_out = 0;
         dime_out   = 0;
         dispense   = 0;
         collect    = 0;

         if ( reset ) D = IDLE;
         else begin
            D = Q;

            case ( Q )
            IDLE:
               if (nickel_in)       D = FIVE;
               else if (dime_in)    D = TEN;
               else if (quarter_in) D = TWENTY_FIVE;
            FIVE:
               if(nickel_in)        D = TEN;
               else if (dime_in)    D = FIFTEEN;
               else if (quarter_in) D = THIRTY;
            TEN:
               if (nickel_in)       D = FIFTEEN;
               else if (dime_in)    D = TWENTY;
               else if (quarter_in) `vend_a_drink;
            TWENTY_FIVE:
               if( nickel_in)       D = THIRTY;
               else if (dime_in)   `vend_a_drink;
               else if (quarter_in) begin

                  `vend_a_drink;
                   nickel_out = 1;
                   dime_out = 1;
               end

            FIFTEEN:
               if (nickel_in)       D = TWENTY;
               else if (dime_in)    D = TWENTY_FIVE;
               else if (quarter_in) begin
                    `vend_a_drink;
                   nickel_out = 1;
               end

            THIRTY:
               if (nickel_in)       `vend_a_drink;
```

```
                else if (dime_in)    begin
                    `vend_a_drink;
                    nickel_out = 1;
                end
                else if (quarter_in) begin
                    `vend_a_drink;
                    dime_out = 1;
                    D = OWE_DIME;
                end

            TWENTY:
                if (nickel_in)       D = TWENTY_FIVE;
                else if (dime_in)    D = THIRTY;
                else if (quarter_in) begin
                    `vend_a_drink;
                    dime_out = 1;
                end

            OWE_DIME:
                begin
                    dime_out = 1;
                    D = IDLE;
                end
            endcase
     end
    end : combinational

    always_ff @ (posedge clk )
       begin : sequential
         Q <= D;
       end : sequential

endmodule : drink_machine
```

Matching block names usage in an interface is shown in Example 8-38. Here, the interface I, module test, and always_comb block all have matching block names.

Example 8-38 Using Matching Block Names in Interfaces

```
interface I;
logic [31:0] i;
logic [31:0] o;
modport MP(input i, output o);
endinterface : I

module test ( I.MP a ) ;
  always_comb
      begin: loop_iterations
            for(int iter = 0; iter <32; iter++)
            a.o[iter] = a.i[iter] ;
      end : loop_iterations
endmodule : test
```

Multidimensional Arrays

In SystemVerilog, multidimensional arrays can be used in

- Multidimensional arrays as function argument types (Example 8-39)
- Module ports as unpacked arrays (Example 8-40, Example 8-41, and Example 8-42)
- Array slicing (Example 8-43)
- Part-select operations (Example 8-44)

For synthesis restrictions on multidimensional arrays, see Appendix B, “Unsupported Constructs.”

The code in Example 8-40 generates and checks the parity of 10 packets. It uses an unpacked array of unpacked structures to model all the bits of all the packets and uses a multidimensional array on module ports.

Example 8-39 Multidimensional Array as Function Argument Type

```
function logic foo ( input logic [10:1][2:1] packed_mda,
                     input logic [10:1] unpacked_mda [2:1] );
//...
endfunction
```

Example 8-40 Module Ports as Unpacked Arrays

```
// Generates and checks parity of ten packets. Uses unpacked array of
unpacked
// structures to model all the bits of all the packets.

typedef struct {
  logic [7:0] hdr1;
  logic [7:0] hdr2;
  logic null_flag;
  logic [27:0] data_body;
} network_packet;

module packet_op_array #(parameter NUM_PACKETS = 10)
   ( input network_packet packet1 [NUM_PACKETS -1:0],
     input network_packet packet2 [NUM_PACKETS -1:0],
     output logic packet_parity1 [NUM_PACKETS -1:0],
     output logic packet_parity2 [NUM_PACKETS -1:0],
   output logic packets_are_equal [NUM_PACKETS -1:0]);

function logic parity_gen (network_packet packet);
  return( ^{packet.hdr1, packet.hdr2, packet.null_flag, packet.data_body}
);
endfunction

function logic compare_packets(network_packet packet1, packet2);
```

```
  if((packet1.hdr1 == packet2.hdr1)
   && (packet1.hdr2 == packet2.hdr2)
   && (packet1.null_flag == packet2.null_flag)
   && (packet1.data_body == packet2.data_body)
  ) return (1'b1);
  else
    return (1'b0);
endfunction

always_comb

begin
    for(int i = 0; i< NUM_PACKETS; i++)
   begin
      packet_parity1[i] = parity_gen(packet1[i]);
      packet_parity2[i] = parity_gen(packet2[i]);
      packets_are_equal[i] = compare_packets(packet1[i], packet2[i]);end

end
endmodule
```

The code in Example 8-41 uses the array query functions $low and $high for loop iteration with the new SystemVerilog for loop syntax; it also has an unpacked array on module ports.

Example 8-41 Module Ports as Unpacked Arrays Using $low and $high

```
module  mda_array_query (input [7:0] a,  output logic t [0:3][0:7], logic
z);
  integer k;
  always_comb
    begin
     for (int j = $low(a, 1) ; j <= $high(a, 1); j++)
      begin
        t[0][j] = a[j];
      end
     for (int i = 1; i < 4; i++)
      begin
        k = 1 << (3-i);
       for (int j = 0; j < k; j++)
         begin
           t[i][j] = t[i-1][2*j] ^ t[i-1][2*j+1];
         end
      end
  end
  assign z = t[3][0];
endmodule
```

The code in Example 8-42 uses unpacked arrays in module ports, the $left and $right array querying functions, and an enhanced for loop to synthesize matrix adders.

Example 8-42 Module Ports as Unpacked Arrays Using $left and $right

```
function automatic logic signed [32:0] add_val1_and_val2
         (logic signed [31:0] val1, val2);
    return (val1 + val2);
endfunction

module matrix_adder (input logic signed [31:0]  a[0:2][0:2],
                     logic signed [31:0] b[0:2][0:2],
                     output logic signed [32:0] sum[0:2][0:2]);
always_comb
  begin
    for (int i=$left(a, 1); i<=$right(a, 1); i++)
      begin
      for (int j=$left(a, 2); j<=$right(a, 2); j++)
         begin
              sum[i][j] = add_val1_and_val2(a[i][j], b[i][j]);
         end
      end
  end
endmodule
```

The code in Example 8-43 slices a multidimensional array into two slices.

Example 8-43 Multidimensional Array Slicing

```
module mda_slicing(input logic[31:0] j[7:0], output int k [1:0]);
  assign k = j[7:6];
endmodule:mda_slicing
```

Example 8-44 uses generate and a part-select operator to assign values to r_val, b_val, and g_val from pixel_array. All are multidimensional arrays and used in module ports.

Example 8-44 Using Part-Select Operations

```
typedef  logic [0:23] three_byte;
typedef logic [0:7] one_byte;

module mda_unpacked_psel (input three_byte pixel_array[0:3],
                          output one_byte r_val[0:3], one_byte
g_val[0:3],
                          one_byte b_val[0:3]);

   genvar i;
   generate
     for ( i=0; i<4; i++) begin: outer_loop

     assign r_val[i] = pixel_array[i][0+:8]; // select all red from
entire array
     assign g_val[i] = pixel_array[i][8+:8]; // select all green from
entire array
     assign b_val[i] = pixel_array[i][16+:8]; // select all blue from
entire array
```

```
   end
   endgenerate

endmodule:mda_unpacked_psel
```

Port Renaming in Multidimensional Arrays

SystemVerilog allows multidimensional arrays in module ports. The names of the module ports are modified in the netlist. To understand the renaming convention, consider Example 8-45.

Example 8-45 Port Renaming in Multidimensional Arrays

```
typedef  logic [0:5] all_component;
typedef logic [0:1] one_component;

module mda_renaming (input all_component pixel_array[0:1],
                      output one_component r_val[0:1], one_component
g_val[0:1],
                     one_component b_val[0:1]);
   genvar i;
     generate
      for ( i=0; i<2; i++) begin: outer_loop

       assign r_val[i] = pixel_array[i][0+:2]; //select all red from
entire array
        assign g_val[i] = pixel_array[i][2+:2]; //select all green from
entire array
       assign b_val[i] = pixel_array[i][4+:2]; //select all blue from
entire array

     end
   endgenerate

endmodule:mda_renaming
```

The ports in Example 8-45 are renamed in the netlist as shown in Example 8-46.

Example 8-46 Port Renaming Example

```
module mda_renaming ( .pixel_array({\pixel_array[0][0] , \pixel_array[0][1] ,
        \pixel_array[0][2] , \pixel_array[0][3] , \pixel_array[0][4] ,
        \pixel_array[0][5] , \pixel_array[1][0] , \pixel_array[1][1] ,
        \pixel_array[1][2] , \pixel_array[1][3] , \pixel_array[1][4] ,
        \pixel_array[1][5] }), .r_val({\r_val[0][0] , \r_val[0][1] ,
        \r_val[1][0] , \r_val[1][1] }), .g_val({\g_val[0][0] , \g_val[0][1] ,
        \g_val[1][0] , \g_val[1][1] }), .b_val({\b_val[0][0] , \b_val[0][1] ,
        \b_val[1][0] , \b_val[1][1] }) );
  input \pixel_array[0][0] , \pixel_array[0][1] , \pixel_array[0][2] ,
         \pixel_array[0][3] , \pixel_array[0][4] , \pixel_array[0][5] ,
         \pixel_array[1][0] , \pixel_array[1][1] , \pixel_array[1][2] ,
         \pixel_array[1][3] , \pixel_array[1][4] , \pixel_array[1][5] ;
  output \r_val[0][0] , \r_val[0][1] , \r_val[1][0] , \r_val[1][1] ,
```

```
        \g_val[0][0] , \g_val[0][1] , \g_val[1][0] , \g_val[1][1] ,
        \b_val[0][0] , \b_val[0][1] , \b_val[1][0] , \b_val[1][1] ;
  wire  \r_val[0][0] , \r_val[0][1] , \r_val[1][0] , \r_val[1][1] ,
        \g_val[0][0] , \g_val[0][1] , \g_val[1][0] , \g_val[1][1] ,
        \b_val[0][0] , \b_val[0][1] , \b_val[1][0] , \b_val[1][1] ;
  assign \r_val[0][0]  = \pixel_array[0][0] ;
  assign \r_val[0][1]  = \pixel_array[0][1] ;
  assign \r_val[1][0]  = \pixel_array[1][0] ;
  assign \r_val[1][1]  = \pixel_array[1][1] ;
  assign \g_val[0][0]  = \pixel_array[0][2] ;
  assign \g_val[0][1]  = \pixel_array[0][3] ;
  assign \g_val[1][0]  = \pixel_array[1][2] ;
  assign \g_val[1][1]  = \pixel_array[1][3] ;
  assign \b_val[0][0]  = \pixel_array[0][4] ;
  assign \b_val[0][1]  = \pixel_array[0][5] ;
  assign \b_val[1][0]  = \pixel_array[1][4] ;
  assign \b_val[1][1]  = \pixel_array[1][5] ;

endmodule
```

9

Troubleshooting and Problem Solving

The following sections describe troubleshooting and problem-solving guidelines:

Expanding Macros and Process Conditional Directives

Macros and conditional compilation directives enable automation of complex tasks and thus reduce coding time. However, this makes code debugging more difficult because of the added code complexity. To help debug SystemVerilog designs, Design Compiler contains a code-expansion feature that enables you to see an expanded version of your original RTL code. This expanded version processes all conditional compilation directives and expands all macro invocations.

This code expansion feature is enabled when you set the `hdlin_sv_tokens` variable to `true`. The default value is `false`.

This section describes how to write out expanded files and provides usage guidelines. It includes RTL design examples along with the associated expanded output files. These are described in the following sections:

- Functionality and Features of Code Expansion
- Code Expansion - Example One
- Code Expansion - Example Two
- Usage Guidelines
- Limitations

Functionality and Features of Code Expansion

You enable code expansion by setting the `hdlin_sv_tokens` variable to `true`. The default value is `false`. When you set this variable to `true`, the tool writes out expanded files (also known as tokens files). The tokens files are named tokens.1.sv, tokens.2.sv, tokens.3.sv and so forth in the order they are written out. All tokens files are written out in the current run directory.

When enabled, the tool produces the tokens file by capturing the exact token stream seen by the parser after preprocessing. This means that if your code generates an error during parsing, the tool creates a tokens file that would be incomplete, or it could even be empty.

The Presto HDL Compiler tool produces a tokens file so that you can see the stream of tokens seen by the parser before an error was encountered. This information could help you in debugging the source of an error and in fixing it.

The following example shows such a case:

```
`define MSFF(q,i,clk,rst)     \
msf_in_lib q`_reg (.o(q),     \
```

```
                    .clk(clk),\
                    .d(i),    \
                    .rst(rst));

module test (output o1, input i1,clk,rst);
    `MSFF(o1,i1,clk,rst)
endmodule
```

Here, q` in the first line causes an error; the corresponding tokens file is incomplete:

```
`line 6  "err.v" 0

`line 7  "err.v" 0
                        module test (output o1, input i1,clk,rst);
`line 8  "err.v" 0
                             msf_in_lib o1
```

Under default conditions, the tool would not generate any output; it would simply report an error. However, with the `hdlin_sv_tokens` variable set to true, the tool generates the output it sees to help the RTL designer fix the error.

Code Expansion - Example One

This section shows an RTL design example that uses macros, a script that uses `hdlin_sv_tokens` to generate an expanded output file, and the contents of that expanded file. This design contains no errors.

Example 9-1 RTL

```
`ifndef SYNTHESIS
 module my_testbench ();
  /* Testbench goes in here. */
 endmodule
`endif

`ifndef GATES
   module TOP_syn (a,clk, o1);
      input a, clk;
  `ifdef NOT
       output o1;
         o1=(!a);
  `elsif FF
      output logic o1;
      always_ff@(posedge clk)
       o1= a;
  `else
       output o1;
       logic temp;
       assign temp = a;
       assign o1 = temp;
```

```
  `endif
   endmodule
`else
   `include "netlist_wrap.sv"
   `include "compiled_gates.v"
`endif

`define DUT(mod) \
      `ifndef GATES \
         mod``_syn \
      `else \
         mod``_svsim \
      `endif
```

Example 9-2 Script

```
set hdlin_sv_tokens true

analyze -f sverilog ex1.v
```

Example 9-3 Contents of Expanded File

```
`line 1  "ex1.sv" 0
`line 6  "ex1.sv" 0
`line 7  "ex1.sv" 0
                                  module TOP_syn (a,clk, o1);
`line 8  "ex1.sv" 0
                                  input a, clk;
`line 9  "ex1.sv" 0
`line 12  "ex1.sv" 0
`line 17  "ex1.sv" 0
                                  output o1;
`line 18  "ex1.sv" 0
                                  logic temp;
`line 19  "ex1.sv" 0
                                  assign temp = a;
`line 20  "ex1.sv" 0
                                  assign o1 = temp;
`line 22  "ex1.sv" 0
                                  endmodule
```

Code Expansion - Example Two

In this example, a large macro is broken into multiple lines. This feature not only enhances human readability, but also simplifies RTL debugging. This design contains no errors.

Example 9-4 RTL

```
`define mak_reg(q,i,clk,en,rst,rstd) \
 logic i_``q ; \
 logic en_``q ;\
 always_comb   \
    if (rst) i_``q = rstd;  \
    else     i_``q = i;     \
 assign en_``q = rst | en ; \
 my_lat myreg``q (.o(q),    \
                  .clk(clk),\
                  .d(i_``q),\
                  .en(en_``q));

module test(output logic out1,
            input  logic in1,
            input  logic clk, en,rst);
`mak_reg(out1,in1,clk,en,rst,'b0)
endmodule
```

Here, the entire `always_comb` block, `assign` statement, `make_reg` macro, and more are in one single line. If any part of this macro has an error, the compiler will point to that large line which would be hard to debug. To overcome this problem and thus simplify RTL debugging, Presto HDL Compiler breaks large macros into smaller lines; the corresponding tokens file is shown below:

Example 9-5 Tokens File

```
`line 12  "ex2.sv" 0
                                  module test(output logic out1,
`line 13  "ex2.sv" 0
                                  input logic in1,
`line 14  "ex2.sv" 0
                                  input logic clk, en,rst);
`line 15  "ex2.sv" 0
                                  logic i_out1 ;
`line 15  "ex2.sv" 0
                                  logic en_out1 ;
`line 15  "ex2.sv" 0
                                  always_comb
`line 15  "ex2.sv" 0
                                  if (rst) i_out1 = 'b0;
`line 15  "ex2.sv" 0
                                  else i_out1 = in1;
`line 15  "ex2.sv" 0
                                  assign en_out1 = rst | en ;
```

```
`line 15  "ex2.sv" 0
                                    my_lat myregout1 (.o(out1),
`line 15  "ex2.sv" 0
                                    .clk(clk),
`line 15  "ex2.sv" 0
                                    .d(i_out1),
`line 15  "ex2.sv" 0
                                    .en(en_out1));
`line 16  "ex2.sv" 0
                                    endmodule
```

Usage Guidelines

Code expansion usage guidelines are as follows:

- The expanded output is available even if there is an error.
- In the expanded output file, the `` `line `` directive specifies the line number. For more information on `` `line ``, see the IEEE Standard 1364-2005.
- If the Design Compiler reader (that is, the Presto HDL Compiler) can read the original RTL, it can read the expanded file.
- You can write out multiple tokens files in a single Design Compiler session.

Limitations

- The code expansion feature applies only to SystemVerilog; that is, it only works with the `analyze -f sverilog`, `read_file -f sverilog`, and `read_sverilog` commands.
- If an input file is encrypted (including an `` `include `` file), a tokens file cannot be written out.

Troubleshooting Generate Loops

To help debug generate loops, use `$display()` as shown in the example below.

Example 9-6

```
// The `ifdef SYNTHESIS is mandatory.
// The $display() will not effect the netlist, but just causes additional
// messages to be written out
// during Presto elaboration.


module test #(N=32)(output [N-1:0] out, input [N-1:0] in);
```

```
genvar I;

generate

  for (I = $left(out); I >= $right(out); I--) begin:GEN

`ifdef SYNTHESIS

    always $display("Instantiating:  mod GEN[%d].inst ( .out(out[%d]),
.in(in[%d])
)", I, I, I);

`endif

    mod inst( .out(out[I]), .in(in[I]) );

  end:GEN

endgenerate

endmodule:test
```

Reducing Simulation/Synthesis Mismatches

The following sections discuss coding styles that might cause synthesis/simulation mismatches:

- Reducing Case Mismatches
- Tasks Inside an always_comb Block
- State Conversion: 2-State and 4-State

Reducing Case Mismatches

It is recommended that you use the unique and priority keywords instead of the `full_case` and `parallel_case` compiler directives, because this method allows the designer's intent to be clearly expressed to the synthesis tool and therefore helps prevent simulation/ synthesis mismatches.

This section describes the recommended replacements for these compiler directives in the following subsections:

- Replace full_case and parallel_case With unique
- Replace full_case With priority
- Replace parallel_case With unique and a Default

- Using priority With a Default

Table 9-1 shows SystemVerilog and Verilog equivalency for the unique and priority constructs and the `full_case` and `parallel_case` compiler directives.

Table 9-1 SystemVerilog and Verilog Equivalency

SystemVerilog	Verilog equivalent
unique case without default	full_case parallel_case
priority case without default	full_case
unique case with default	parallel_case
priority case with default	no compiler directives

Replace full_case and parallel_case With unique

In SystemVerilog, the unique keyword, attached to a case statement without a default, is the same as the Synopsys `full_case` and `parallel_case` compiler directives. It is recommended that you use the unique keyword instead of the compiler directives, because this method allows the designer's intent to be clearly expressed to the synthesis tool and therefore helps prevent simulation/synthesis mismatches.

In Example 9-7 the unique keyword is used for the case statement.

Example 9-7 SystemVerilog: unique Keyword on case Statement

```
        typedef struct {
        logic a_sel;
        logic b_sel;
        } priority_sel;
module unique_case_without_default_struct(input priority_sel one_hot_sel,
                                          output logic a_hi, logic b_hi);

   always_comb
       unique case (1'b1)
         one_hot_sel.a_sel : begin a_hi = '1; b_hi = '0; end
         one_hot_sel.b_sel : begin a_hi = '0; b_hi = '1; end
       endcase

endmodule
```

Although not recommended, two equivalent ways to write this code using the Synopsys `full_case` and `parallel_case` compiler directives are shown in Example 9-8 and Example 9-9.

Example 9-8 SystemVerilog: full_case/parallel_case

```
            typedef struct {
           logic a_sel;
           logic b_sel;
           } priority_sel;

    module full_case_parallel_case_struct(input priority_sel one_hot_sel,
                                          output logic a_hi, logic b_hi);

       always_comb
          case (1'b1)    //synopsys full_case parallel_case
             one_hot_sel.a_sel : begin a_hi = '1; b_hi = '0; end
             one_hot_sel.b_sel : begin a_hi = '0; b_hi = '1; end
          endcase

    endmodule
```

Example 9-9 Verilog 2001: full_case and parallel_case

```
    module full_case_parallel_case_struct (input a_sel, input b_sel,
                                          output reg a_hi, output reg b_hi);

       always@(*)
           case (1'b1)    //synopsys full_case parallel_case
              a_sel : begin a_hi = 1'b1; b_hi = 1'b0; end
              b_sel : begin a_hi = 1'b0; b_hi = 1'b1; end
           endcase

    endmodule
```

All three coding styles build the same logic, which is shown in Example 9-10.

Example 9-10 Netlist

```
module full_case_parallel_case_struct ( a_sel, b_sel, a_hi, b_hi );
  input a_sel, b_sel;
  output a_hi, b_hi;
  wire   a_hi, b_hi;
  assign a_hi = a_sel;
  assign b_hi = b_sel;
endmodule
```

All three coding styles give the same statistics for the case statement, pointing to the line where the case is in the RTL, as shown:

```
================================================
|           Line           |  full/ parallel   |
================================================
|            9             |    user/user      |
================================================
Presto compilation completed successfully.
```

Important: Mixing both the compiler directive and the keyword, as shown in Example 9-11, creates a VER-517 error. The error message is shown in Example 9-12.

Example 9-11 Mixed Coding Style Creates a VER-517 Error

```
        typedef struct {
         logic a_sel;
         logic b_sel;
         } priority_sel;

module mixed_struct(input priority_sel one_hot_sel,
                    output logic a_hi, logic b_hi);

   always_comb
      unique case (1'b1)     // synopsys full_case parallel_case,
                           // both compiler directive and keyword
         one_hot_sel.a_sel : begin a_hi = '1; b_hi = '0; end
         one_hot_sel.b_sel : begin a_hi = '0; b_hi = '1; end
      endcase

endmodule
```

Example 9-12 VER-517 Error Message

```
Error:  ..../full_case_parallel_case/mixed.sv:9: A  case statement is
marked
with both a unique or priority keyword, and a full_case or parallel_case
directive.
(VER-517)
*** Presto compilation terminated with 1 errors. ***
```

Replace full_case With priority

In SystemVerilog, the priority keyword, attached to a case statement without a default, is the same as the Synopsys `full_case` compiler directive. It is recommended that you use the priority keyword instead of the compiler directive because this method allows the designer's intent to be clearly expressed to the synthesis tool and therefore helps prevent simulation/synthesis mismatches.

In Example 9-13, the priority keyword is used on the case statement.

Example 9-13 SystemVerilog: priority on case Statement With No Default

```
           typedef struct {
           logic a_sel;
           logic b_sel;
           } priority_sel;

    module priority_case_without_default_struct(input priority_sel
    one_hot_sel,
                                                output logic a_hi, logic b_hi);

          always_comb
             priority case (1'b1)
                one_hot_sel.a_sel : begin a_hi = '1; b_hi = '0; end
                one_hot_sel.b_sel : begin a_hi = '0; b_hi = '1; end
             endcase

    endmodule
```

Although not recommended, two equivalent ways to write this code using the Synopsys `full_case` compiler directive are shown in Example 9-14 and Example 9-15.

Example 9-14 shows equivalent code using SystemVerilog constructs and the `full_case` compiler directive.

Example 9-14 SystemVerilog: Using full_case

```
           typedef struct {
           logic a_sel;
           logic b_sel;
           } priority_sel;

     module full_case_struct (input priority_sel one_hot_sel,
                              output logic a_hi, logic b_hi);

          always_comb
              case (1'b1)    //synopsys full_case
                one_hot_sel.a_sel : begin a_hi = '1; b_hi = '0; end
                one_hot_sel.b_sel : begin a_hi = '0; b_hi = '1; end
              endcase

    endmodule
```

Example 9-15 shows equivalent code using Verilog constructs and the full_case compiler directive.

Example 9-15 Verilog 2001: Using full_case

```
module full_case_struct(input a_sel, input b_sel,
                        output reg a_hi, output reg b_hi);

   always@(*)
      case (1'b1)    //synopsys full_case
         a_sel : begin a_hi = 1'b1; b_hi = 1'b0; end
         b_sel : begin a_hi = 1'b0; b_hi = 1'b1; end
      endcase

endmodule
```

All three coding styles build logically equivalent circuits. Example 9-16 shows the netlist for the coding styles using the `full_case` compiler directive.

Example 9-16 Netlist

```
module full_case_struct ( a_sel, b_sel, a_hi, b_hi );
  input a_sel, b_sel;
  output a_hi, b_hi;
  wire   a_hi;
  assign a_hi = a_sel;
  IV U4 ( .A(a_hi), .Z(b_hi) );
endmodule
```

All three coding styles give the same statistics for the case statement, pointing to the line where the case is in the RTL, as shown:

```
================================================
|           Line           |  full/ parallel   |
================================================
|            9             |     user/no       |
================================================
Presto compilation completed successfully.
```

Important:

Mixing both the compiler directive and the priority case construct generates an ELAB-909 warning message, as shown:

```
Warning:  ../rtl/ch9.ex15.sv:4: Case statement is not a full case.
(ELAB-909)
```

Replace parallel_case With unique and a Default

In SystemVerilog, the unique keyword, attached to a case statement with a default, is the same as the Synopsys `parallel_case` compiler directive. It is recommended that you use the keyword instead of the compiler directive, because the keyword allows the designer's intent to be clearly expressed to the synthesis tool and therefore helps prevent simulation/ synthesis mismatches.

In Example 9-17, the unique keyword is used for the case statement.

Example 9-17 SystemVerilog: unique Keyword With Default

```
            typedef struct {
            logic a_sel;
            logic b_sel;
            } priority_sel;

    module unique_case_with_default_struct(input priority_sel one_hot_sel,
                                           output logic a_hi, logic b_hi);

          always_comb
             unique case (1'b1)
                one_hot_sel.a_sel : begin a_hi = '1; b_hi = '0; end
                one_hot_sel.b_sel : begin a_hi = '0; b_hi = '1; end
                default:;
             endcase

    endmodule
```

Although not recommended, two equivalent ways to write this code using the Synopsys `parallel_case` compiler directive are shown Example 9-18 and Example 9-19.

Example 9-18 shows equivalent code using SystemVerilog constructs and the `parallel_case` compiler directive.

Example 9-18 SystemVerilog: parallel_case

```
            typedef struct {
            logic a_sel;
            logic b_sel;
            } priority_sel;

    module parallel_case_struct(input priority_sel one_hot_sel,
                                output logic a_hi, logic b_hi);

          always_comb
             case (1'b1)    //synopsys parallel_case
                one_hot_sel.a_sel : begin a_hi = '1; b_hi = '0; end
                one_hot_sel.b_sel : begin a_hi = '0; b_hi = '1; end
             endcase

    endmodule
```

Example 9-19 shows equivalent code using Verilog constructs and the `parallel_case` compiler directive.

Example 9-19 Verilog 2001: parallel_case

```
module parallel_case_struct(input a_sel, input b_sel, output reg a_hi,
output
reg b_hi);

    always@(*)
       case (1'b1)    //synopsys parallel_case
          a_sel : begin a_hi = 1'b1; b_hi = 1'b0; end
          b_sel : begin a_hi = 1'b0; b_hi = 1'b1; end
       endcase

endmodule
```

All three coding styles build the same logic which is shown in Example 9-20.

Example 9-20 Netlist

```
module parallel_case_struct ( a_sel, b_sel, a_hi, b_hi );
  input a_sel, b_sel;
  output a_hi, b_hi;
  wire   N0;
  LD1 b_hi_reg ( .G(N0), .D(b_sel), .Q(b_hi) );
  LD1 a_hi_reg ( .G(N0), .D(a_sel), .Q(a_hi) );
  OR2 U3 ( .A(a_sel), .B(b_sel), .Z(N0) );
endmodule
```

All three coding styles give the same statistics for the case statement as shown below:

Example 9-21 case Statistics

```
================================================
|           Line            |  full/ parallel  |
================================================
|            9              |     no/user      |
================================================
```

Important:
Mixing both the compiler directive and the unique keyword generates an ELAB-910 warning message, as shown:

```
Warning:  ../rtl/ch9.ex19.sv:4: Case statement is not a parallel case.
(ELAB-910)
```

Using priority With a Default

In SystemVerilog, using the priority keyword, attached to a case statement with a default, is the same as not specifying any Synopsys `full_case` or `parallel_case` compiler directives to the synthesis tool.

In Example 9-22 the priority keyword and a default are used with the case statement.

Example 9-22 SystemVerilog: priority case and Default

```
          typedef struct {
          logic a_sel;
          logic b_sel;
          } priority_sel;

    module priority_case_with_default_case_struct(input priority_sel
    one_hot_sel,
                                                  output logic a_hi, logic
    b_hi);

         always_comb
              priority case (1'b1)
                   one_hot_sel.a_sel : begin a_hi = '1; b_hi = '0; end
                   one_hot_sel.b_sel : begin a_hi = '0; b_hi = '1; end
                   default:;
               endcase
    endmodule
```

Equivalent ways to write this code are shown in Example 9-23 and Example 9-24. Example 9-23 uses SystemVerilog while Example 9-24 uses Verilog 2001.

Example 9-23 SystemVerilog: No Compiler Directives

```
          typedef struct {
          logic a_sel;
          logic b_sel;
          } priority_sel;
    module no_pragma_case_struct(input priority_sel one_hot_sel,
                                 output logic a_hi, logic b_hi);

         always_comb
            case (1'b1)
               one_hot_sel.a_sel : begin a_hi = '1; b_hi = '0; end
               one_hot_sel.b_sel : begin a_hi = '0; b_hi = '1; end
            endcase

    endmodule
```

Example 9-24 shows equivalent code using Verilog constructs.

Example 9-24 Verilog 2001: No Compiler Directives

```
    module no_pragam_case_struct(input a_sel, input b_sel,
                                 output reg a_hi, output reg b_hi);

       always@(*)
          case (1'b1)
             a_sel : begin a_hi = 1'b1; b_hi = 1'b0; end
             b_sel : begin a_hi = 1'b0; b_hi = 1'b1; end
          endcase

    endmodule
```

All three coding styles build the same logic, which is shown in Example 9-25.

Example 9-25 Netlist

```
module no_pragam_case_struct ( a_sel, b_sel, a_hi, b_hi );
  input a_sel, b_sel;
  output a_hi, b_hi;
  wire   N0, N1, n4;
  LD1 b_hi_reg ( .G(N0), .D(N1), .Q(b_hi) );
  LD1 a_hi_reg ( .G(N0), .D(a_sel), .Q(a_hi) );
  NR2 U5 ( .A(a_sel), .B(n4), .Z(N1) );
  IV U6 ( .A(b_sel), .Z(n4) );
  OR2 U7 ( .A(b_sel), .B(a_sel), .Z(N0) );
endmodule
```

All three coding styles give the same statistics for the case statement, which is shown below.

Example 9-26 case Statistics

```
============================================
|           Line           |  full/ parallel  |
============================================
|            9            |      no/no       |
============================================
```

Tasks Inside an always_comb Block

The LRM does not define the behavior of tasks inside always_comb blocks, but it does define the behavior for functions. To avoid a simulation/synthesis mismatch, use void functions instead of tasks inside always_comb blocks.

To understand why the mismatch can happen, consider what the IEEE Std 1800-2005 says:

“always_comb is sensitive to changes within the contents of a function, whereas always @ is only sensitive to changes to the arguments of a function.”*

Although the LRM does not say it is illegal to use tasks inside always_comb blocks, it does not specify how always_comb blocks should behave with tasks inside the sensitivity list. This could cause a simulation and synthesis mismatch.

To illustrate this mismatch, consider the following:

- The code in Example 9-27, which uses a task inside the always_comb block
- The accompanying testbench (Example 9-28), GETCH netlist (Example 9-29), and simulation log (Example 9-30)

Example 9-27 RTL With Task In an always_comb Block

```
module comb1(input logic a, b ,c, output logic [1:0] y);

   always_comb orf1(a);

      function void orf1 (a);
         y[0] = a | b | c;
      endfunction

   always_comb ort1 (a);

      task ort1 (a);
         y[1] = a | b | c;
      endtask

endmodule
```

Example 9-28 shows the testbench for Example 9-27.

Example 9-28 Testbench for Design in Example 9-27

```
module comb1_tb(output logic a, b, c);
initial
   begin
 a = 0; b = 0; c = 0;
 #10 a = 0; b = 0; c = 1;
 #10 a = 0; b = 1; c = 0;
 #10 a = 0; b = 1; c = 1;
 #10 a = 1; b = 0; c = 0;
 #10 a = 1; b = 0; c = 1;
 #10 a = 1; b = 1; c = 0;
 #10 a = 1; b = 1; c = 1;
   end
endmodule

module top;
wire a_w, b_w, c_w;
wire y1_w, y0_w ;

comb1 u1(a_w, b_w, c_w, {y1_w, y0_w});
comb1_tb u2(a_w, b_w, c_w);

initial
   begin
 $display("\t\tTime A B C Y1 Y0\n");
 $monitor($time,,,,a_w,,,,b_w,,,,c_w,,,,y1_w,,,,y0_w);
    end
endmodule
```

Example 9-29 shows the GTECH netlist for Example 9-27.

Example 9-29 GTECH Netlist

```
module comb1 ( a, b, c, y );
  output [1:0] y;
  input a, b, c;
  wire   N0, N1;
  GTECH_OR2 C7 ( .A(N0), .B(c), .Z(y[0]) );
  GTECH_OR2 C8 ( .A(a), .B(b), .Z(N0) );
  GTECH_OR2 C9 ( .A(N1), .B(c), .Z(y[1]) );
  GTECH_OR2 C10 ( .A(a), .B(b), .Z(N1) );
endmodule
```

Example 9-30 shows the simulation log for Example 9-27.

Example 9-30 Simulation Log

```
Chronologic VCS simulator copyright 1991-2008
Contains Synopsys proprietary information.
Compiler version B-2008.12; Runtime version B-2008.12;  Mar 26 06:34 2009
                Time   A   B   C  Y1  Y0
                   0   0   0   0   0   0
                  10   0   0   1   0   1
                  20   0   1   0   0   1
                  30   0   1   1   0   1
                  40   1   0   0   1   1
                  50   1   0   1   1   1
                  60   1   1   0   1   1
                  70   1   1   1   1   1
           V C S   S i m u l a t i o n   R e p o r t
Time: 70
CPU Time:      0.020 seconds;       Data structure size:   0.0Mb
Thu Mar 26 06:34:11 2009
```

In synthesis, the tool produces four 2-input OR gates, as shown in Example 9-29. The outputs y[0] and y[1] are both outputs of 2-input OR gates. The simulation log (Example 9-30), shows

- y[0] going to 1 when any of the inputs is 1, which is correct. Note that y[0] is the output from the void function call orf1.
- y[1] going to 1 only when input A goes to 1, and it is not sensitive to the changes of B and C. This is incorrect behavior that occurs because y[1] is the output of task ort1 inside an always_comb block.

This situation results in a simulation/synthesis mismatch, and the synthesis tool issues the following VER-520 warning to indicate that a task is used inside an always_comb block:

```
Running PRESTO HDLC
Compiling source file …/comb.1.sv
Warning:  …/comb.1.sv:6: Task enable in always_comb block. (VER-520)
```

To avoid this simulation/synthesis mismatch, use void functions inside always_comb blocks, as shown in Example 9-31, and do not use tasks.

Example 9-31 Recommended Coding Style

```
module comb1(input logic a, b ,c, output logic [1:0] y);

   always_comb orf1(a);

      function void orf1 (a);
         y[0] = a | b | c;
         y[1] = a | b | c;
      endfunction

endmodule
```

State Conversion: 2-State and 4-State

The tool treats 2-state variables the same as 4-state variables. (See Appendix B, “Unsupported Constructs.”) Therefore a simulation/synthesis mismatch can occur when you convert from 4-state to 2-state or from 2-state to 4-state. The simulation tool considers an “x” value as a “don’t know” value, whereas the synthesis tool considers an “x” value as a “don’t care” value.

In Example 9-32, “a” is an input of logic type (4-state) making a continuous assignment to “b” which is bit type (2-state). The testbench drives “a” through the variable “a_driver” which is also logic type but is uninitialized at time 0. Therefore for simulation, this is a “don’t know” situation, as shown in the simulation log in Example 9-33.

Because you have the statement, assign b = a, and b is bit type, its default value is 0 (if uninitialized), and therefore the simulation log has A = x and B = 0 at time 0.

To avoid this simulation/synthesis mismatch, use only 2-state or only 4-state variables and avoid such conversions.

Example 9-32 RTL

```
module logic_bit_test(input logic a, output bit b);
   assign b = a;
endmodule

module logic_bit_testbench(output logic a_driver);
   initial begin // no intial value
      #10 a_driver = '1;
      #10 a_driver = '0;
      #10 $finish;
   end
endmodule

module top;
   wire a_con, b_con;

   logic_bit_test u1(a_con, b_con);
   logic_bit_testbench u2(a_con);

   initial
     begin
      $display("\t\tTime A  B\n");
      $monitor($time,,,,a_con,,,,b_con);
    end
endmodule
```

Example 9-33 Simulation Log

```
Chronologic VCS simulator copyright 1991-2008
Contains Synopsys proprietary information.
Compiler version B-2008.12; Runtime version B-2008.12;  Mar 26 07:46 2009
                Time   A   B
                   0   x   0
                  10   1   1
                  20   0   0
$finish called from file
 "redu.sim.syn.mismatch_state.conver.2state.4state.sv", line 8.
$finish at simulation time                   30
           V C S   S i m u l a t i o n   R e p o r t
Time: 30
CPU Time:      0.020 seconds;       Data structure size:   0.0Mb
Thu Mar 26 07:46:11 2009
```

Detecting Unintended Hardware and Empty Blocks

When you use the always_ff, always_latch, and always_comb constructs, the tool expects certain hardware. If the expected hardware is not inferred, or if the block might be removed during compile, the tool generates a warning message. For details, see

- “Using always_comb and Inferring a Register” on page 4-2
- “Using always_comb with Empty Blocks” on page 4-3
- “Using always_latch and Not Inferring a Sequential Device” on page 5-3
- “Using always_latch With Empty Blocks” on page 5-4
- “Using always_ff and Not Inferring a Sequential Device” on page 5-5
- “Using always_ff With Empty Blocks” on page 5-6

Port Connection (.*) Reading Restriction

When using the .* port connection style, you must analyze all the lower-level modules before elaborating the top-level module; otherwise the tool gives an ELAB-397 error message.

To understand this, consider the following two files:

```
// top.sv has the following code:

module top(input logic in, output logic out);
bottom b1(.*);
endmodule

// bottom.sv has the following code:

module bottom(input logic in, output logic out);
assign out = in;
endmodule
```

If you analyze the top-level module and elaborate the top design without analyzing the bottom, as in the following script,

```
analyze -f sverilog top.sv
elaborate top
```

you will see the following error:

```
Error:  ./top.sv:2: The module or interface 'bottom' needs to be
analyzed. (ELAB-397)
```

To prevent this error, perform either one of the following tasks:

- Modify the script to analyze the bottom module before you elaborate the top.

```
analyze -f sverilog {bottom.sv top.sv}
elaborate top
```

- Provide a placeholder bottom module in the same file as top.sv, as shown in the following example:

```
module bottom(input logic in, output logic out);
// this is a placeholder module that contains port names but no content
endmodule
```

Using do...while Loops

A do....while loop is synthesizable if the exit condition is deterministic. The tool does not handle unknown initial values in loops when the number of iterations can still be bounded. However, the simulator tool, VCS, does not have this restriction. For example, Example 9-34 is synthesizable; Example 9-35 is not. In Example 9-35, the exit condition "x" is an unknown initial value; therefore the tool reports a VER-1010 error.

Example 9-34 Synthesizable do...while Loop

```
module do_while_test2(input logic [3:0] count1, output logic [3:0] z);
  logic [3:0] x, count;
  always_comb
  begin
    x = 4'd2;
    count = count1;
    do
    begin
      count++;
      x++;
    end
    while(x < 4'd15);
  z = count;
  end
endmodule
```

Example 9-35 Unsynthesizable do...while Loop

```
module do_while_test2(input logic [3:0] count1,  output logic[3:0] z);
  logic [3:0]  count, x;
  always_comb
  begin
    count = count1;
    do
    begin
      count++;
      x++;
    end
    while(x < 4'd15);
    z = count;
  end
endmodule
```

Using the typedef Construct

You need to define typedef before using it. Example 9-36 shows an allowed typedef; in Example 9-37, the typedef is not allowed.

Example 9-36 Typedef Allowed

```
   typedef logic mytype;

   module allowed(input logic clock, input mytype in, output mytype out);
      always_ff@(posedge clock)
            out = in;
 endmodule
```

Example 9-37 Typedef Not Allowed

```
 module not_allowed(input logic clock, input mytype in, output mytype
out);

     always_ff@(posedge clock)
           out <= in;
 endmodule

 typedef logic mytype;
```

Reading Assertions in Synthesis

The following SystemVerilog keywords are parsed and ignored: assert, assume (VCS, the simulation tool, may not support this keyword at this time), before, bind, bins, binsof, class, clocking, constraint, cover, coverpoint, covergroup, cross, endclass, endclocking, endgroup, endpackage, endprogram, endproperty, endsequence, extends, final, first_match, intersect, ignore_bins, illegal_bins, local, package, program, property, protected, sequence, super, this, var, throughout, within. If an assertion-related keyword is not parsed and ignored, it is considered to be unsupported. For these unsupported keywords, see “Unsupported Constructs” in Appendix B.

Example 9-38 shows how the synthesis tool parses and ignores the “assert” keyword. The example correctly infers a flip-flop, as shown in the inference report in Example 9-39.

Example 9-38

```
module dff_with_imm_assert(input DATA, CLK, RESET, output logic
Q);
   //synopsys sync_set_reset "RESET"
  always_ff @(posedge CLK)
    if (~RESET)
        begin
         Q <= 1'b0;
             assert (Q == 1'b0)
             $display("%m PASS:Flip Flop got reset");
             else
             $display("%m FAIL:Flip Flop got reset");
        end
    else
         Q <= DATA;

 endmodule
```

Example 9-39 Inference Report

```
================================================================================
|    Register Name    |   Type    | Width | Bus | MB | AR | AS | SR | SS | ST |
================================================================================
|        Q_reg        | Flip-flop |   1   |  N  | N  | N  | N  | Y  | N  | N  |
================================================================================
Presto compilation completed successfully.
```

Other Troubleshooting Guidelines

- For unsupported SystemVerilog constructs, see Appendix B, “Unsupported Constructs.
- If you are having problems with $unit, make sure you are following the coding guidelines in Chapter 2, “Global Name Space ($unit).
- If you are having problems with interfaces, make sure you are following the coding guidelines in Chapter 7, “Interfaces.
- Regarding casting, the use of nonvoid function calls as statements is supported but generates a warning.
- In some cases, the tool does not correctly rename the output. See “Renaming Example 3” on page 7-26.
- If your design contains interfaces, you cannot use the elaborate command to instantiate a parameterized design. See “Reading SystemVerilog Files” on page 1-6.
- If your design uses checker libraries, see “Reading Designs With Assertion Checker Libraries” on page 1-22.
- Generally, all the restrictions for HDL Compiler (Presto Verilog) apply to the SystemVerilog tool. For details, see the *HDL Compiler (Presto Verilog) Reference Manual*.

A

SystemVerilog Design Examples

This appendix contains the following two examples that build FIFOs using various SystemVerilog constructs:

- FIFO Example 1
- FIFO Example 2

FIFO Example 1

Example A-1 uses many SystemVerilog features such as structure, typedef, different types of module ports, always_comb, always_ff, and unpacked array of structures to build a FIFO.

Example A-1 FIFO 1

```
// Synchronous FIFO. 4 x 16 bit words.
//
typedef logic [7:0] ubyte;

typedef struct {
    ubyte src;
    ubyte dst;
    ubyte [0:3] data;
  } packet_t;

module fifo #(DEPTH=2, MAX_COUNT = (1<<DEPTH)) ( input clk, input rstp,
 input packet_t din, input readp, input writep, output packet_t dout,
 output logic emptyp, output logic fullp);

// Define the FIFO pointers. A FIFO is essentially a circular queue.
//
reg [(DEPTH-1):0] tail;
reg [(DEPTH-1):0] head;

// Define the FIFO counter. Counts the number of entries in the FIFO which
// is how we figure out things like Empty and Full.
//
reg [DEPTH:0] count;

// Define our regsiter bank.
// ***************************
// *** Array of structures ***
// ***************************
packet_t fifomem[0:MAX_COUNT];

// Dout is registered and gets the value that tail points to
always_ff @(posedge clk) begin
  if (rstp == 1)
    dout <= '{default:0};
  else
    dout <= fifomem[tail];
end

// Update FIFO memory.
always_ff @(posedge clk) begin
  if (rstp == 1'b0 && writep == 1'b1 && fullp == 1'b0)
    fifomem[head] <= din;
end

// Update the head register.
//
always_ff @(posedge clk) begin
  if (rstp == 1'b1)
    head <= 0;
  else
```

```
    if (writep == 1'b1 && fullp == 1'b0)
      // WRITE
      head <= head + 1;
end

// Update the tail register.
//
always_ff @(posedge clk) begin
  if (rstp == 1'b1)
    tail <= 0;
  else
    if (readp == 1'b1 && emptyp == 1'b0)
      // READ
      tail <= tail + 1;
end

// Update the count regsiter.
//
always_ff @(posedge clk) begin
  if (rstp == 1'b1) begin
    count <= 0;
  end
  else begin
    case ({readp, writep})
      2'b00:  count <= count;
      2'b01:
              // WRITE
              if (!fullp)
                count <= count + 1;
      2'b10:
              // READ
              if (!emptyp)
                count <= count - 1;
      2'b11:
              // Concurrent read and write.. no change in count
              count <= count;

    endcase
  end
end

// Update the flags
//
always_comb begin
  if (count == 0)
    emptyp = 1'b1;
  else
    emptyp = 1'b0;
end

// Update the full flag
//
always_comb begin
  if (count < MAX_COUNT)
    fullp = 1'b0;
  else
    fullp = 1'b1;
end
```

```
endmodule
```

FIFO Example 2

Example A-2 uses many SystemVerilog features such as structure, typedef, interface with modport, different types of module ports, always_comb, always_ff, and unpacked array of structures to build a FIFO.

Example A-2 FIFO 2

```
// Synchronous FIFO. 4 x 16 bit words.
//
typedef logic [7:0] ubyte;

typedef struct {
    ubyte src;
    ubyte dst;
    ubyte [0:3] data;
} packet_t;

// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// Use interface & modport to declare data in / out
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
interface port;
  logic enable;
  logic stall;

  packet_t packet;
  modport sendm(input enable, packet, output stall);
  modport recvm(input enable, output packet, stall);
endinterface : port

module fifo #(DEPTH = 2, MAX_COUNT = (1<<DEPTH)) ( input clk, input rstp, port.sendm
 in, port.recvm out);
// Define the FIFO pointers. A FIFO is essentially a circular queue.
//
reg [(DEPTH-1):0] tail;
reg [(DEPTH-1):0] head;

// Define the FIFO counter. Counts the number of entries in the FIFO which
// is how we figure out things like Empty and Full.
//
reg [(DEPTH):0] count;

// Define our register bank.
//
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// Array of structures
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
packet_t fifomem[0:MAX_COUNT];

// Dout is registered and gets the value that tail points to RIGHT NOW.
always_ff @(posedge clk) begin
  if (rstp == 1)
```

```
    out.packet <= '{default:0};
  else
    out.packet <= fifomem[tail];
end

// Update FIFO memory.
always_ff @(posedge clk) begin
  if (rstp == 1'b0 && in.enable == 1'b1 && in.stall == 1'b0)
    fifomem[head] <= in.packet;
end

// Update the head register.
//
always_ff @(posedge clk) begin
  if (rstp == 1'b1)
    head <= 0;
  else
    if (in.enable == 1'b1 && in.stall == 1'b0)
      // WRITE
      head <= head + 1;
end

// Update the tail register.
//
always_ff @(posedge clk) begin
  if (rstp == 1'b1)
    tail <= 0;
  else
    if (out.enable == 1'b1 && out.stall == 1'b0)
      // READ
      tail <= tail + 1;
end

// Update the count regsiter.
//
always_ff @(posedge clk) begin
  if (rstp == 1'b1) begin
    count <= 0;
  end
  else begin
    case ({out.enable, in.enable})
    2'b00: count <= count;
    2'b01:
            // WRITE
            if (!in.stall)
              count <= count + 1;
    2'b10:
            // READ
            if (!out.stall)
              count <= count - 1;
    2'b11:
            // Concurrent read and write.. no change in count
            count <= count;
    endcase
  end
end

// First, update the empty flag.
```

```
//
always_comb begin
  if (count == 0)
    out.stall = 1'b1;
  else
    out.stall = 1'b0;
end

// Update the full flag
//
always_comb begin
  if (count < MAX_COUNT)
    in.stall = 1'b0;
  else
    in.stall = 1'b1;
  end
endmodule : fifo
```

B

Unsupported Constructs

The Synopsys SystemVerilog tool does not support all the synthesis features described in the SystemVerilog LRM. Generally, all the restrictions for HDL Compiler (Presto Verilog) apply to the SystemVerilog tool. For details, see the HDL Compiler (Presto Verilog) Reference Manual.

This appendix includes the following section:

Unsupported SystemVerilog Constructs

The following constructs are unsupported:

1. Automatic variables in static tasks and functions are not supported.
2. Clocking blocks, defined by a clocking-endclocking keyword pair, are not supported in synthesis. They are parsed and ignored.
3. The following SystemVerilog keywords are parsed and ignored: assert, assume, before, bind, bins, binsof, class, clocking, constraint, cover, coverpoint, covergroup, cross, endclass, endclocking, endgroup, endpackage, endprogram, endproperty, endsequence, extends, final, first_match, intersect, ignore_bins, illegal_bins, local, package, program, property, protected, sequence, super, this, var, throughout, within.
4. System functions $onehot, $onehot0, $countones, and $isunknown are not supported.
5. Non-ANSI style interface ports are not supported.

   The following code is not supported:

   ```
   module M (i, o, sb)
     input i;
     output o;
   simple_bus sb;
   //simple_bus is an interface port (not an instantiation)
   ```

   For this code, the tool returns the following error message:

   ```
   Error: nonansi.v:2: Syntax error at or near token 'input'.
   (VER-294)
   ```

6. Compiler directives (such as operator label) in interfaces are not supported.

   The following code is not supported:

   ```
   a = b + /* synopsys label my_adder */ c;
   ```

7. Attributes are not fully supported; they are treated as comments.

   ```
   (* comment *)
   is same as
   // comment
   ```

8. Ternary operators for struct literals are not supported.
9. 2-state values are not supported; they are treated as 4-state values. This can cause simulation/synthesis mismatches. For details, see “State Conversion: 2-State and 4-State” on page 9-19. Per the LRM, int, bit, shortint, byte, and longint are 2-state data types with legal values of 0 and 1. Static variables of 2-state data types without explicit

declaration initializations are initialized to 0 instead of x. Design Compiler initializes all static variables to x (including those with explicit declaration initializations), even if they are of 2-state data types.

10. Autoassignments as expressions are not supported.

    The following code is not supported:

    ```
    mask & ( in << i++ )
    ```

    When the tool reads this code

    ```
    module m (input a, output b);
    int i;
    assign b = a + i++;
    endmodule
    ```

    it reports the following error message:

    ```
    Error:  i1.v:3: The construct "assignment expression" is
    not supported.  (VER-721)
    ```

11. Casting on types: $cast is not supported.

12. Variable initialization is ignored in synthesis.

13. Struct literals from SystemVerilog 3.0: Ternary operators for struct literals are not supported.

14. Arrayed instantiation of interfaces are not supported.

15. Generic interfaces: A module with a generic interface port cannot be the top module for elaboration. This coding style is not supported.

16. Variables referred to across an interface-type port are readable and writable. Therefore, the semantics of access are like those of a module ‘ref’ port. In computer memory, this is similar to a call by reference, where the last write wins. But in real hardware, this can only happen when modeled by the resolution semantics of wires. Because of this, ref ports are not realizable in silicon hardware, and hence not synthesizable. They are used only in simulation.

17. The following SystemVerilog keywords are not supported:
    alias, chandle, context, dist (accepted only in testbenches), expect, export, extern, foreach, new, null, pure, shortreal, static, solve, string, tagged, virtual, wait_order, with.

    If you use an unsupported keyword, the tool reports an error and terminates. To prevent this error, wrap the construct as follows:

    ```
    `ifndef SYNTHESIS
          ...
    `endif
    ```

18. The following keywords are not supported: forkjoin, join_any, join_none, rand, randc, ref, randcase, randsequence.

19. timeunit, timeprecision statements, and the delay are ignored for synthesis.

20. real/time ports on modules/interfaces/tasks/functions and real/time multidimensional arrays are not supported.

21. Unpacked unions are not supported.

    The following code is not supported:

```
22.  union {
   logic       my_logic;
   logic      [63:0] my_logic;
   logic    [63:0] my_logic;
   longint        my_longint;
        } a;
```

23. Forward declarations of typedefs are not supported.

    The following code is not allowed because you must indicate what you typedef “foo” to in the same line.

```
typedef foo;
foo p;
typedef int foo;
```

24. The following code is not supported:

```
25.  module latch (output logic [31:0] y, input [31:0] a,
              input enable);
  always @(a iff enable == 1)
  y = a;
endmodule
```

26. Nested module declarations are not supported.

C

New Feature and Enhancement Summary

This appendix summarizes the new features and enhancements in the SystemVerilog synthesis tool. It includes the following section:

- New SystemVerilog Features and Enhancements

New SystemVerilog Features and Enhancements

Table C-1 New SystemVerilog Features and Enhancements

As of the C-2009.06 release	Feature description
Hierarchical elaboration flow	For details, see “Hierarchical Elaboration Flow for SystemVerilog Designs With Interfaces” on page 1-9.
Automatic detection of RTL language from file extensions by using the `read_file -format` command.	For details, see “Automatic Detection of RTL Language From File Extensions” on page 1-7.
As of the B-2008.09 release	**Feature description**
Assignment patterns	For details, see “Assignment Patterns” on page 8-3.
analyze -vcs	For details, see “Reading Large Designs” on page 1-8.
Enumerated types methods	For details, see “Methods for Enumerated Types” on page 6-8.
Wildcard equality and inequality operators	For syntax, see the SystemVerilog LRM.
As of the A-2007.12 release	**Feature description**
Debugging enhancement.	For details, see “Expanding Macros and Process Conditional Directives” on page 9-2.
Support for packages. Scope extraction using: Wildcard inports inside modules Wildcard imports inside $unit Specific imports	For details, see “Packages” in Chapter 3.
Support for acs_read_hdl.	For details, see “Reading SystemVerilog With acs_read_hdl” on page 1-8.
Support for automatic variable initialization.	For details, see “Automatic Variable Initialization” on page 8-6.
Support for wildcard import.	For details, see “Wildcard Import into $unit” on page 3-6.
Support for binding tasks and functions by .name.	For details, see “Tasks and Functions - Binding by .name” on page 8-5.
Support for ranges of enum labels.	For details, see “Specify enum Ranges” on page 6-8.

Table C-1 New SystemVerilog Features and Enhancements

Support for begin_keyword and end_keyword.	For details, see "'begin_keywords and 'end_keywords" on page 8-4.
As of the Z-2007.03 release	Feature description
SystemVerilog netlist wrapper	This wrapper enables a SystemVerilog design instance to be driven by the testbench. See "Testbenches - Using the Netlist Wrapper" on page 1-31.
Global declaration support using $unit.	See "$unit Usage" on page 2-2
As of the Y-2006.06 release	Feature description
Cross-tool compatibility: shortening long names	The tool allows you to shorten object names to enable cross-tool compatibility. See "Naming Problems: Long Names" on page 1-21.
As of the X-2005.09 release	Feature description
No new tool features or enhancements.	Examples in the manual were updated to show additional SystemVerilog construct usage.
As of the W-2004.12 release	Feature description
Enhanced for loop support	See "Variables in for Loops" on page 8-6."
Bit-level support for `one_hot` and `one_cold` compiler directives	See "Bit-Level Support for Compiler Directives" on page 8-7.
Checker library support	See "Reading Designs With Assertion Checker Libraries" on page 1-22.
Warnings for empty always_comb, always_ff, and always_latch blocks	See "Using always_comb with Empty Blocks" on page 4-3, "Using always_ff With Empty Blocks" on page 5-6, and "Using always_latch With Empty Blocks" on page 5-4.
As of the V-2004.06 release	Feature description
Constant declarations in $unit	Constant declarations are supported in $unit. See "$unit Usage" on page 2-2.
Implicit port connection (".*")	The .* port connection style is supported. See "Implicit Instantiation of Ports" on page 8-14.
Assertions	Assertion-related keywords are parsed and ignored. See Appendix B, "Unsupported Constructs."

Table C-1 New SystemVerilog Features and Enhancements

Enforced checking for automatic functions and tasks in interfaces and $unit	See "$unit Synthesis Restrictions" on page 2-10 and "Interface Restrictions" on page 7-27.
Bit-level support for compiler directives	See "Bit-Level Support for Compiler Directives" on page 8-7.

Index

Symbols

Numerics

A

B

C

D

E

F

O

P

R

S

T

U